Resilient web design

by Jeremy Keith

Resilient Web Design was written and produced by Jeremy Keith, and typeset in ET Book.

Licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

RESILIENT WEB DESIGN

Introduction

With a title like

Resilient Web Design, you might think that this is a handbook for designing robust websites. This is not a handbook. It's more like a history book.

Marshall McLuhan once said:

> *We look at the present through a rear-view mirror. We march backwards into the future.*

But in the world of web design, we are mostly preoccupied with the here and now. When we think beyond our present moment, it is usually to contemplate the future—to imagine the devices, features, and interfaces that don't yet exist. We don't have time to look back upon our past, and yet the history of web design is filled with interesting ideas.

The World Wide Web has been around for long enough now that we can begin to evaluate the twists and turns of its evolution. I wrote this book to highlight some of the approaches to web design that have proven to be resilient. I didn't do this purely out of historical interest (although I am fascinated by the already rich history of our young industry). In learning from the past, I believe we can better prepare for the future.

You won't find any code in here to help you build better websites. But you will find ideas and approaches. Ideas are more resilient than code. I've tried to combine the most resilient ideas from the history of web design into an approach for building the websites of the future.

I hope you will join me in building a web that lasts; a web that's resilient.

RESILIENT WEB DESIGN

CHAPTER 1:

Foundations

THE HISTORY OF HUMAN civilisation is a tale of cumulative effort. Each generation builds upon the work of their forebears. Sometimes the work takes a backward step. Sometimes we wander down dead ends. But we struggle on. Bit by bit our species makes progress. Whether the progress is incremental or a huge leap forward, it is always borne upon the accomplishments of those who came before us.

Nowhere is this layered nature of progress more apparent than in the history of technology. Even the most dramatic bounds in technological advancement are only possible when there is some groundwork to build upon.

Gutenberg's printing press would not have been invented if it weren't for the work already done in creating the screw press used for winemaking. Technologies aren't created in isolation. They are imprinted with the ghosts of their past.

The layout of the QWERTY keyboard for your computer—and its software equivalent on your phone—is an echo of the design of the first generation of typewriters. That arrangement of keys was chosen to reduce the chances of mechanical pieces of metal clashing as they sprang forward to leave their mark on the paper.

The hands on a clock face move in a clockwise direction only because that's the direction that the shadow cast by a sundial moves over the course of a day in the northern hemisphere. Had history turned out differently, with the civilisation of the southern hemisphere in the ascendent, then the hands on our clocks would today move in the opposite direction. As for why those clocks carve out the time in periods of 24 hours, each with 60 minutes, with each minute having 60 seconds, that's thanks to an ancient Sumerian civilisation. They hit upon using the number 60 as their base for counting and commerce. It's the lowest number that can be equally divided by the first six numbers. That sexagesimal way of counting is still with us today in the hours, minutes, and seconds that we use as conceptual models for subdividing one rotation of our planet.

The shadow cast by a sundial in the southern hemisphere moves counter-clockwise.

These echoes of the past reverberate in the present even when their usefulness has been outlived. You'll still sometimes see a user interface that displays an icon of a Compact Disc or vinyl record to represent music. That same interface might use the image of a 3½ inch floppy disk to represent the concept of saving data. The reason why floppy disks wound up being 3½ inches in size is because the disk was designed to fit into a shirt pocket. The icons in our software interfaces are whispering stories to us from the history of clothing and fashion.

Let's share what we know

Scientific progress would be impossible without a shared history of learning to build upon. As Sir Isaac Newton put it, if we have seen further it is by standing on the shoulders of giants.

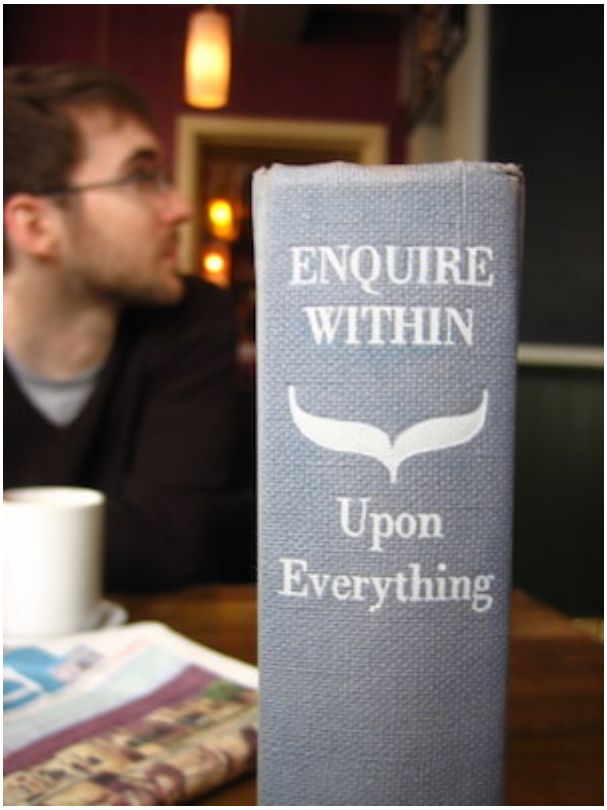
When knowledge is passed from one generation to the next, theories become more refined, units of measurement become standardised, and experiments increase in precision.

Right now humanity's most precise experiments are being conducted beneath the border between Switzerland and France. This is the home of CERN, the European Organisation for Nuclear Research. In the 16-mile wide ring of its Large Hadron Collider, protons are being smashed together at velocities approaching the speed of light. Our primate species is recreating the conditions from the start of our universe. The LHC is the most complex piece of machinery that has ever been built.

The awe-inspiring engineering of the LHC is matched by the unprecedented levels of international co-operation behind CERN. The particle accelerator became operational in the first decade of the 21st century but the groundwork was laid more than half a century before. That was when a group of nations came together to create the CERN Convention, dedicating resources and money towards pure scientific research. The only return on investment they expected was in the currency of knowledge.

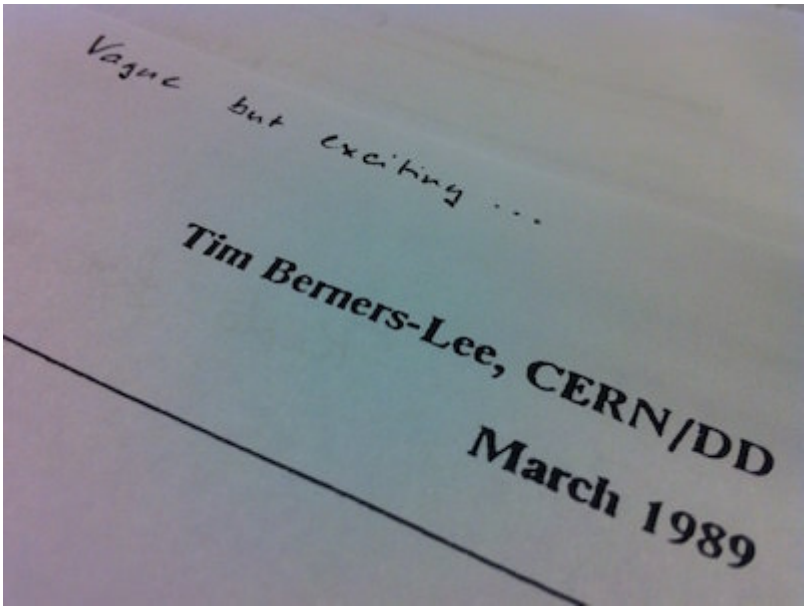
This groundwork created a unique environment free from the constraints of national, economic, and social hierarchies. Nobel prize-winning physicists collaborate with students on summer internships. If there is an element of social categorisation at CERN, it is only between theorists and experimentalists. The theorists are the ones with blackboards in their offices. The experimentalists are the ones with computers. They have to deal with a lot of data. Even before the Large Hadron Collider was switched on, managing information was a real challenge at CERN.

Enter Tim Berners-Lee, a computer scientist from England who found himself working at CERN in the 1980s. At the beginning of that decade, he started a personal project to get to grips with managing information. The resulting software was called ENQUIRE, named for a Victorian manual of domestic life called *Enquire Within Upon Everything*.



Enquire Within Upon Everything

By the end of the ’80s, Tim Berners-Lee was ready to tackle the thorny problem of information management on a larger scale. In order to get buy-in at CERN, he produced an unassuming document with the title *Information Management: A Proposal*. Fortunately his supervisor, Mike Sendall, recognised the potential of the idea and gave the go-ahead by scrawling the words “vague but exciting” across the top of the paper. That proposal would become the World Wide Web.



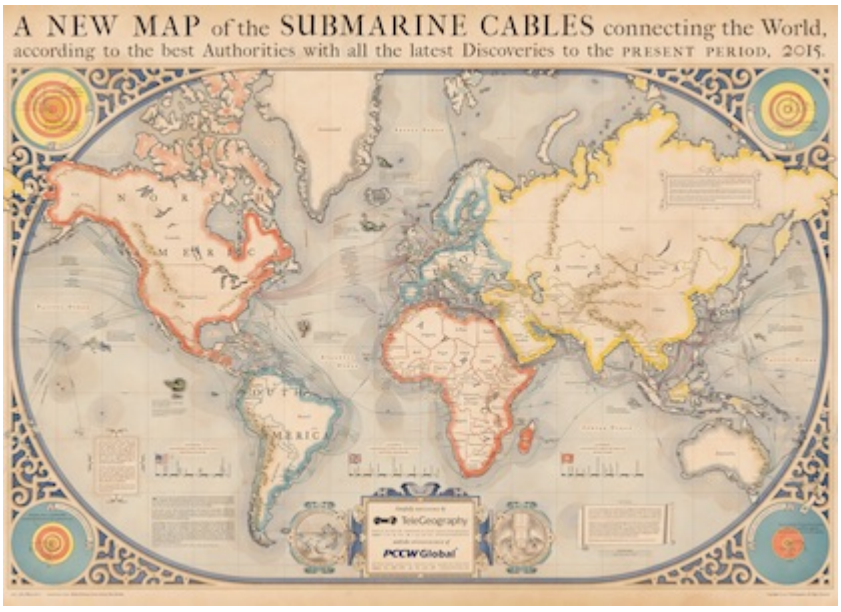
Vague but exciting...

Net value

Today we think of the World Wide Web as one of the greatest inventions in the history of communication, but to the scientists at CERN it is merely a byproduct. When you're dealing in cosmological timescales and investigating the very building blocks of reality itself, the timeline of humanity's relationship with technology is little more than a rounding error.

When Tim Berners-Lee decided to take on the problem of information management at CERN, the internet was already established as part of the infrastructure there. This network of networks was first created in the 1960s and the early adopters were universities and scientific institutions.

These nodes were already physically connected via telephone wires. Rather than build an entirely new network from scratch, it made sense to reuse what was already there. Once again, a new technology was only made possible by the existence of an older one. In the nineteenth century the world was technologically terraformed for the telegraph. Through astonishing feats of engineering, our planet was wired up with undersea cables. Those same cables would later be repurposed to carry telephone signals. Later again, they would carry the digital ones and zeros of the internet. Today those signals are transmitted via pulses of light travelling through fibre-optic cables. Those fibre-optic cables follow the same paths across the ocean floor as their telegraphic antecedents.


A NEW MAP of the SUBMARINE CABLES connecting the World,
according to the best Authorities with all the latest Discoveries to the PRESENT PERIOD, 2015.
TeleGeography
PCCW Global

Telegeography publish a new map of undersea cables every year.

The internet has no centre. This architectural decision gives the network its robustness. You may have heard that the internet was designed to resist a nuclear attack. That's not entirely correct. It's true that the project began with military considerations. The initial research was funded by DARPA, the Defense Advanced Research Projects Agency. But the engineers working on the project were not military personnel. Their ideals had more in common with the free-speech movement than with the military-industrial complex. They designed the network to route around damage, but the damage they were concerned with was censorship, not a nuclear attack.

The open architecture of the internet reflected the liberal worldview of its creators. As well as being decentralised, the internet was also deliberately designed to be a dumb network. That's its secret sauce. The protocols underlying the transmission of data on the internet—TCP/IP—describe how packets of data should be moved around, but those protocols care not a whit for the contents of the packets. That allows the internet to be the transport mechanism for all sorts of applications: email, Telnet, FTP, and eventually the World Wide Web.

Hyperspace

The web uses HTTP, the HyperText Transfer Protocol, to send and receive data. This data is uniquely identified with a URL. Many of these URLs identify pages made of HTML, the HyperText Markup Language. The killer feature of the web lies here in HTML's humble A element. The A stands for Anchor. Its HREF attribute allows you to cast off from within one URL out to another URL, creating a link that can be traversed from one page to the other. These links turn the web from being a straightforward storage and retrieval system into a hypertext system.

Tim Berners-Lee did not invent hypertext. That term was coined by Ted Nelson, a visionary computer scientist who was working on his own hypertext system called Xanadu. Both Ted Nelson and Tim Berners-Lee were influenced by the ideas set out by Vannevar Bush in his seminal 1945 essay, *As We May Think*. Bush, no doubt, was in turn influenced by the ideas of Belgian informatician Paul Otlet. Each one of these giants in the history of hypertext was standing on the shoulders of the giants that had come before them. Giants all the way down.

Compared to previous grand visions of hypertext, links on the web are almost laughably simplistic. There is no two-way linking. If you link to another URL, but the page at that URL is moved or destroyed, you have no way of knowing.

But the simplicity of the web turned out to be the secret of its success.

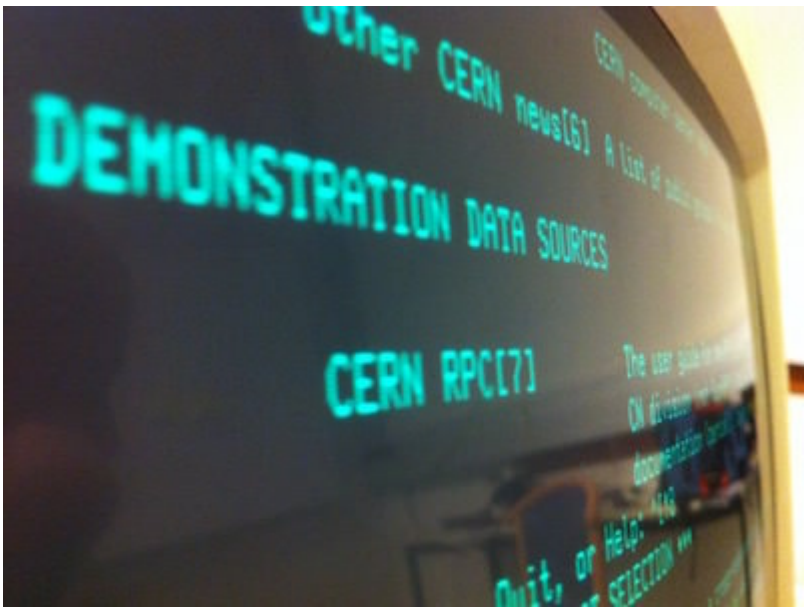
Tim Berners-Lee assumed that most URLs would point to non-HTML resources; word-processing documents, spreadsheets, and all sorts of other computer files. HTML could then be used to create simple index pages that point to these files using links. Because HTML didn't need to do very much, it had a limited vocabulary. That made it relatively easy to learn. To Tim Berners-Lee's surprise, people began to create fully-fledged documents in HTML. Instead of creating content in other file formats and using HTML to link them up, people began writing content directly in HTML.

Mark me up, mark me down

HTML wasn't the first markup language to be used at CERN. Scientists there were already sharing documents written in a flavour of SGML—Standard Generalized Markup Language. Tim Berners-Lee took this existing vocabulary from CERN SGML and used it as a starting point for his new markup language. Once again, it made sense to build on what people were already familiar with rather than creating something from scratch.

The first version of HTML contained a grand total of 21 elements. Many of those elements are still with us today —TITLE, P, UL, LI, H1, H2, etc., and of course, the A element. Others have fallen by the wayside—ISINDEX, PLAINTEXT, LISTING, HP1, HP2, etc., as well as a proprietary element called NEXTID that only made sense if you were using a computer running the NeXTSTEP operating system. That's the OS that Tim Berners-Lee was using when he created HTTP, HTML, and the world's first web browser, called confusingly WorldWideWeb, which only worked on NeXT machines.

To demonstrate the power and interoperability of the web, a cross-platform browser was needed; one that anybody could install and use, no matter what operating system they were using. The task of building that browser fell to an undergraduate at CERN named Nicola Pellow. She created the Line Mode Browser. It was simple but powerful. It didn't have the same level of interactivity as the WorldWideWeb browser, but the fact that it could be run on any machine meant that the web was now accessible to everyone.



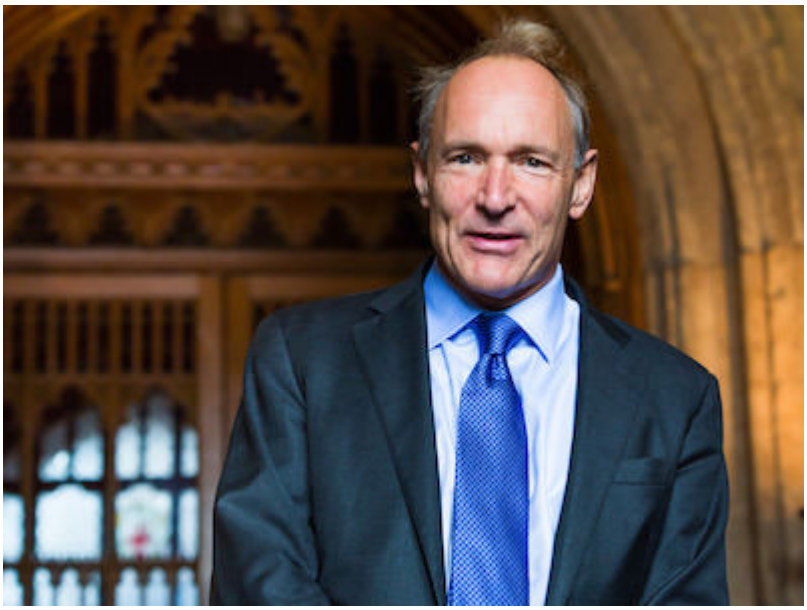
The Line Mode Browser in action.

As soon as there were two web browsers in the world, interoperability and backwards compatibility became important issues. For instance, what should the Line Mode Browser do when it encounters an HTML element it doesn't understand, such as NEXTID?

The answer can be found in the sparse documentation that Tim Berners-Lee wrote for his initial collection called *HTML Tags*. Under the heading "Next ID" he wrote:

> *Browser software may ignore this tag.*

This seemingly innocuous decision would have far-reaching consequences for the future of the World Wide Web.



Sir Tim Berners-Lee.
Photograph by Paul Clark.
Licensed under a Creative Commons Attribution-ShareAlike 4.0 International license.

RESILIENT WEB DESIGN

CHAPTER 2:

Materials

At the risk of teaching

grandmother to suck eggs, I'd like you to think about what happens when a browser parses an HTML element. Take, for example, a paragraph element with some text inside it. There's an opening P tag, a closing P tag, and between those tags, there's the text.

```
<p>some text</p>
```

A web browser encountering this element will display the text between the opening and closing tags. Now consider what happens when that same web browser encounters an element it doesn't recognise.

```
<marklar>some more text</marklar>
```

Once again, the browser displays the text between the opening and closing tags. What's interesting here is what the browser doesn't do. The browser does not throw an error. The browser does not stop parsing the HTML at this point, refusing to go any further. Instead, it simply ignores the tags and displays the content within.

This liberal attitude to errors allowed the vocabulary of HTML to grow over time from the original 21 elements to the 121 elements in HTML5. Whenever a new element is introduced to HTML, we know exactly how older browsers will treat it; they will ignore the tags and display the content.

That's a remarkably powerful feature. It allows browsers to implement new HTML features at different rates. We don't have to wait for every browser to recognise a new element. Instead we can start using the new element at any time, secure in the knowledge than non-supporting browsers won't choke on it.

```
<main>this text will display in any
browser</main>
```

If web browsers treat all tags the same way—displaying their contents—then what's the point of having a vocabulary of elements in HTML?

The meaning of markup

Some HTML elements are literally meaningless. The SPAN element says nothing about the contents within it. As far as a web browser is concerned, you may as well use a non-existent MARKLAR element. But that's the exception. Most HTML elements exist for a reason. They have been created and agreed upon in order to account for specific situations that authors like you and I are likely to encounter.

There are obviously special elements, like the A element, that come bundled with superpowers. In the case of the A element, its superpower lies in the HREF attribute that allows us to link out to any other resource on the web. Other elements like INPUT, SELECT, TEXTAREA, and BUTTON have their own superpowers, allowing people to enter data and submit it to a web server.

Then there are elements that describe the kind of content they contain. The contents of a P element should be considered a paragraph of text. The contents of an LI element should be considered as an item in a list. Browsers display the contents of these elements with some visual hints as to their meaning. Paragraphs are displayed with whitespace before and after their content. List items are displayed with bullet points or numbers before their content.

The early growth of HTML's vocabulary was filled with new elements that provided visual instructions to web browsers: BIG, SMALL, CENTER, FONT. In fact, the visual instructions were the only reason for those elements to exist—they provided no hint as to the *meaning* of the content they contained. HTML was in danger of becoming a visual instruction language instead of a vocabulary of meaning.

A matter of style

Håkon Wium Lie was working at CERN at the same time as Tim Berners-Lee. He immediately recognised the potential of the World Wide Web and its language, HTML. He also realised that the expressive power of the language was in danger of being swamped by visual features. Lie proposed a new format to describe the presentation of HTML documents:
Cascading Style Sheets.

He was quickly joined by the Dutch programmer Bert Bos. Together they set about creating a syntax that would be powerful enough to handle the demands of designers, while remaining simple enough to learn quickly. They succeeded.

Think for a moment of all the sites out there on the web. There's a huge variation in visual style: colour schemes, typographic treatments, textures and layouts. All of that variety is made possible by one simple pattern that describes all the CSS ever written:

```
selector {
    property: value;
}
```

That's it.

CSS shares HTML's forgiving attitude to errors. If a web browser encounters a selector it doesn't understand, it simply skips over whatever is between that selector's curly braces. If a browser sees a property or a value it doesn't understand, it just ignores that particular declaration. The browser does not throw an error. The browser does not stop parsing the CSS at this point, refusing to go any further.

```
marklar {
    marklar: marklar;
}
```

Just as with HTML, this loose error-handling has allowed CSS to grow over time. New selectors, new properties, and new values have been added to the language's vocabulary over the years. Whenever a new feature lands in CSS, designers and developers know that they can safely use it, even if it isn't yet widely supported in browsers. They can rest assured that old browsers will react to new features with complete indifference.

Just because a language is elegant and well-designed doesn't mean that people will use it. CSS arrived later than HTML. Designers didn't spend the intervening years waiting patiently for a way to style their documents on the web. They used what was available to them.

Killing it

In 1996 David Siegel published a book entitled *Creating Killer Websites*. In it, he outlined a series of ingenious techniques for wrangling eye-catching designs out of the raw material of HTML.

One technique involved using a transparent GIF, just one pixel by one pixel in size. If this was inserted into a page as an IMG element, but given precise values in its WIDTH and HEIGHT attributes, designers could control the amount of whitespace in their designs.

Another technique used the TABLE element. This element—along with its children TR and TD—was intended to describe tabular data. But with the right values applied to the widths and heights of table cells, it could be used to recreate just about any desired layout.

These were hacks; clever solutions to tricky problems. But they had unfortunate consequences. Designers were treating HTML as a tool for the appearance of content instead of a language for describing the meaning of content. CSS was a solution to this problem, if only designers could be convinced to use it.

Browser wars

One of the reasons why web designers weren't using CSS was the lack of browser support. Back then there were two major browsers competing for the soul of the web: Microsoft Internet Explorer and Netscape Navigator. They were incompatible by design. One browser would invent a new HTML element or attribute. The other browser would invent their own separate element or attribute to do exactly the same thing.

Perhaps the thinking behind this strategy was that web designers would have to choose which proprietary features they were going to get behind, like children being forced to choose between parents. In reality web designers had little choice but to write for both browsers which meant doing twice the work.

A group of web designers decided enough was enough. They gathered together under the banner of the Web Standards Project and began lobbying Microsoft and Netscape to abandon their proprietary ways and adopt standards such as CSS.

The tide began to turn with the launch of Internet Explorer 5 for the Mac, a browser that shipped with impressive CSS support. If this was the future of web design, life was about to get a lot more productive and creative.

Some forward-thinking web designers caught the CSS bug early. They redesigned their websites using CSS for layout instead of using TABLEs and spacer GIFs. True to the founding spirit of the web, they shared what they were learning and encouraged others to make the switch to CSS.

Perhaps the best demonstration of the power of CSS was a website called the CSS Zen Garden, created by Dave Shea. It was a showcase of beautiful and varied designs, all of them accomplished with CSS. Crucially, the HTML remained the same.

A selection from the cornucopia of designs created for the CSS Zen Garden.

Seeing the same HTML document styled in a multitude of different ways drove home one of the beneficial effects of CSS: separation of concerns.

Coupling

In any system, from urban infrastructure to a computer program, the designers of that system can choose the degree to which the pieces of the system depend on one another. In a tightly-coupled system, every piece depends on every piece. In a loosely-coupled system, all the pieces are independent, with little to no knowledge of the other pieces.

In a tightly-coupled system, each part of the system can make assumptions about the other parts. These systems can be designed quite quickly, but at a price. They lack resilience. If one piece fails, it could take the whole system with it.

Designing a loosely-coupled system can take more work. The payoff is that the overall result is more resilient to failure. Individual parts of the system can be swapped out with a minimum of knock-on effects.

The hacks pioneered by David Siegel tightly coupled structure and presentation into a single monolithic HTML file. The adoption of CSS eased this dependency, bringing the web closer to the modular approach of the UNIX philosophy. The presentational information could be moved into a separate file: a style sheet. That's how a single HTML document at the CSS Zen Garden could have so many different designs applied to it.

The style sheet still needs to have some knowledge of the HTML document's structure. Quite often, "hooks" are added into the markup to make it easier to style: specific values of CLASS or ID attributes, for example. So HTML and CSS aren't completely decoupled. They form a partnership but they also have an arrangement. The markup document might decide that it wants to try seeing other style sheets sometimes. Meanwhile, the style sheet could potentially be applied to other documents. They are loosely coupled.

Dancing about architecture

It takes time for a discipline to develop its own design values. Web design is a young discipline indeed. While we slowly begin to form our own set of guiding principles, we can look to other disciplines for inspiration.

The world of architecture has accrued its own set of design values over the years. One of those values is the principle of material honesty. One material should not be used as a substitute for another. Otherwise the end result is deceptive.

Using TABLEs for layout is materially dishonest. The TABLE element is intended for marking up the structure of tabular data. The end result of using TABLEs, FONT elements, and spacer GIFs is a façade. At first glance everything looks fine, but it won't stand up to scrutiny. As soon as such a website is stress-tested by actual usage across a range of browsers, the façade crumbles.

Using CSS for presentation is materially honest—that's the intended use of CSS. It also allows HTML to be materially honest. Instead of trying to fulfil two roles—structure and presentation—HTML can return to fulfilling its true purpose, marking up the meaning of content.

It's still possible to use (or abuse) CSS to be materially dishonest. For the longest time, there was no easy way to add rounded corners to an element using CSS. Instead, web designers found ways to hack around the problem, putting background images on the element to simulate the same end effect. It worked up to a point, but just like the spacer GIF hack, it was a façade. Then the `border-radius` property arrived. Now designers can have their rounded corners in a materially honest way.

Crucially, designers were able to use new properties like `border-radius` long before every web browser supported them. That's all thanks to the liberal error-handling model of CSS. Newer browsers would display the rounded corners. Older browsers would not throw an error. Older browsers would not stop parsing the CSS and refuse to parse any further. They would simply ignore the instructions they didn't understand and move on. No harm, no foul.

Of course this means that the resulting website will look different in different browsers. Some people will see rounded corners. Others won't.

And that's okay.

RESILIENT WEB DESIGN

CHAPTER 3:

Visions

Design adds clarity. Using colour, typography, hierarchy, contrast, and all the other tools at their disposal, designers can take an unordered jumble of information and turn it into something that's easy to use and pleasurable to behold. Like life itself, design can win a small victory against the entropy of the universe, creating pockets of order from the raw materials of chaos.

The Book of Kells is a beautifully illustrated manuscript created over 1200 years ago. It's tempting to call it a work of art, but it is a work of design. The purpose of the book is to communicate a message; the gospels of the Christian religion. Through the use of illustration and calligraphy, that message is conveyed in an inviting context, making it pleasing to behold.

The incipit to the Gospel of Matthew in the Book of Kells.

Design works within constraints. The Columban monks who crafted the Book of Kells worked with four inks on vellum, a material made of calfskin. The materials were simple but clearly defined. The cenobitic designers knew the hues of the inks, the weight of the vellum, and crucially, they knew the dimensions of each page.

Prints and the revolution

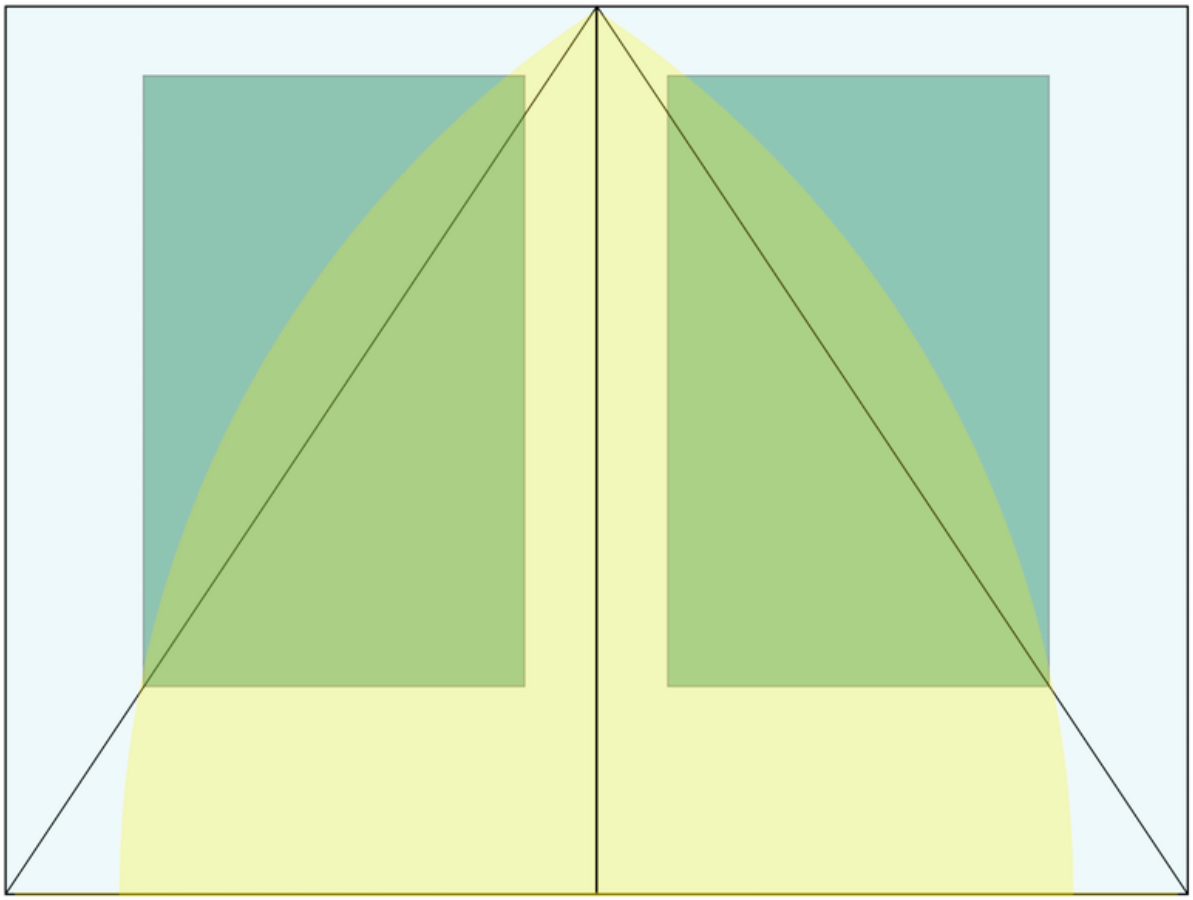
Materials and processes have changed and evolved over the past millennium or so. Gutenberg's invention of movable type was a revolution in production. Whereas it would have taken just as long to create a second copy of the Book of Kells as it took to create the first, multiple copies of the Gutenberg bible could be produced with much less labour. Even so, many of the design patterns such as drop caps and columns were carried over from illuminated manuscripts. The fundamental design process remained the same: knowing the width and height of the page, designers created a pleasing arrangement of elements.

Incipit epistola sancti ihe[ro]nimi ad paulinum presbiterum de omnibus diuine hystorie libris. capitulū p̄mū.

Frater ambrosius tua michi munuscula p̄ferens: deuulit sil et suauissimas ītās: q̄ a principio amicicie: fide: pbitate: et iam fide ītās et veteris amicicie noua: p̄ferebant. Vbi era cū illa necessitudo ē: et xp̄i gluino copulata: q̄m non vilitas rei familiaris: nō p̄cia tantum corpoꝛis: nō lubrica et palpās adulatio: sed dei amoꝛ: et diuinaꝛ scripturaꝝ studia conciliant. Legim⁹ in veteribꝫ hystoꝛijs: quosdā lustrasse p̄uincias: nouos adisse p̄los: maria trāsisse: ut eos quos ex libris nouerant: corā q̄s videāt. Sicut pitagoras emphiticos uates: sic plato egipti: et architā tarentinū: eudoxusq; oram ytalie: que quondā magna grecia dicebat: laboriosissime petagrit: et ut qui achemis mḡr erat: et poteꝛis: auiusq; doctrinas achademie gignasia p̄fessabat: fieret pegnus atq; discipulus: malēs aliena uerecūde discere: q̄m sua ip̄udenter ingerē. Democrit⁹ cū trās quasi toto orbe fugientes p̄sequit⁹ capt⁹ a piratis et venūdatus: etiāno cuidā dominꝰ paruit: ductꝰ captiuus uinctꝰ et seruus. Tamē quia p̄fus maioꝝ emerere se fuit: ad ortum fluuiū: lacus eloquētie fonte manantē: de ulāmis hispanie galliarūq; finibꝫ quosdam uenisse nobiles legimus: et quos ad exemplaꝝ sui roma nō traxerat: unꝰ hoīs fama p̄duxit. Habuit illa etas inaudita oīnibꝫ seculis celebrandūq; miraculū: ut urbē tantā ingressi: aliud extra urbem quererent. Apollonius siue ille magꝰ ut vulgꝰ loquitur: siue p̄fus: ut pitagoꝛici tradunt: intrauit p̄sas: p̄terijt caucasū: albanos: scithas: massagetas: opulentissima indie regna penetrauit: et ad extremum tandem phrson ampne trānisso pueuit ad bragmanas: ut hyarcam in chrono sedentē aureo et de tantali fonte potantem: inter pauꝯ discipulos: de natura: de moribꝫ: ac de cursu diuꝝ et side: audiret docentem. Inde p̄ elamitas: babilonios: chaldeos: medos: assyrios: parthos: syros: phenices: arabes: palestinos: rursus ad alexandriā: peruenit ad ethiopiā: ut gymnosophistas et famosissimam solis mensam videret in sabulo. Inuenit ille vir ubiq; q̄ discere: et semp proficies: semp se melioꝝ fieret. Scripsit super hoc planissime octo voluminibus: phylostratus.

Quid loquar de ecclī hominibꝫ: cū ap̄lus paulus: vas electionis: et magister gentiū: qui de conscientia trāi se hospitis loquebat: dicēs. An experimentū queritis eius qui in me loquit⁹ xp̄c. Post damascū arabiāq; lustratā: ascendit ihersolimā ut videret petrū et mansit apud eū diebꝫ quindecim. Hoc cū nuncio ebdomadis et ogdoadis: futuꝝ gentiū p̄dicatoꝝ instruendus erat. Curcūq; post annos quatuordecim assumpto barnaba et tito: egressus est cū ap̄lis euāgeliū: ne forte in vacuum curreret aut cucurrisset. Habet ecclā q̄d lacerius energie: viue voces actus: et in aures discipli de auctoris ore transfusa: fortius sonāt. Vnde et eschines cū rodi exularet: et legeret

The techniques of the print designer reached their zenith in the 20th century with the rise of the Swiss Style. Its structured layout and clear typography is exemplified in the work of designers like Josef Müller-Brockmann and Jan Tschichold. They formulated grid systems and typographic scales based on the preceding centuries of design.



A framework for medieval manuscripts by Jan Tschichold.

Knowing the ratio of the dimensions of a page, designers could position elements with maximum effect. The page is a constraint and the grid system is a way of imposing order on it.

Taking your talent to the web

When the web began to conquer the world in the 1990s, designers started migrating from paper to pixels. David Siegel's *Creating Killer Websites* came along at just the right time. Its clever TABLE and GIF hacks allowed designers to replicate the same kind of layouts that they had previously created for the printed page.

Those TABLE layouts later became CSS layouts, but the fundamental thinking remained the same: the browser window—like the page before it—was treated as a known constraint upon which designers imposed order.

There's a problem with this approach. Whereas a piece of paper or vellum has a fixed ratio, a browser window could be any size. There's no way for a web designer to know in advance what size any particular person's browser window will be.

Designers had grown accustomed to knowing the dimensions of the rectangles they were designing within. The web removed that constraint.

If it ain't fixed, don't break it

There's nothing quite as frightening as the unknown. These words of former US Secretary of Defense Donald Rumsfeld should be truly terrifying (although the general consensus at the time was that they sounded like nonsense):

> *There are known knowns. There are things we know we know. We also know there are known unknowns, that is to say we know there are some things we do not know. But there are also unknown unknowns—the ones we don't know we don't know.*

The ratio of the browser window is just one example of a known unknown on the web. The simplest way to deal with this situation is to use flexible units for layout: percentages rather than pixels. Instead, designers chose to pretend that the browser dimensions were a known known. They created fixed-width layouts for one specific window size.

In the early days of the web, most monitors were 640 pixels wide. Web designers created layouts that were 640 pixels wide. As more and more people began using monitors that were 800 pixels wide, more and more designers began creating 800 pixel wide layouts. A few years later, that became 1024 pixels. At some point web designers settled on the magic number of 960 pixels as the ideal width.

It was as though the web design community were participating in a shared consensual hallucination. Rather than acknowledge the flexible nature of the browser window, they chose to settle on one set width as the ideal …even if that meant changing the ideal every few years.

Not everyone went along with this web-wide memo.

Dao or dao not

In the year 2000 the online magazine A List Apart published an article entitled *A Dao of Web Design*. It has stood the test of time remarkably well.

In the article, John Allsopp points out that new mediums often start out by taking on the tropes of a previous medium. Scott McCloud makes the same point in his book *Understanding Comics*:

> *Each new medium begins its life by imitating its predecessors. Many early movies were like filmed stage plays; much early television was like radio with pictures or reduced movies.*

With that in mind, it's hardly surprising that web design began with attempts to recreate the kinds of layouts that designers were familiar with from the print world. As John put it:

> *"Killer Web Sites" are usually those which tame the wildness of the web, constraining pages as if they were made of paper – Desktop Publishing for the Web.*

Web design can benefit from the centuries of learning that have informed print design. Massimo Vignelli, whose work epitomises the Swiss Style, begins his famous Canon with a list of The Intangibles including discipline, appropriateness, timelessness, responsibility, and more. Everything in that list can be applied to designing for the web. Vignelli's Canon also includes a list of The Tangibles. That list begins with paper sizes.

The web is not print. The known constraints of paper—its width and height—simply don't exist. The web isn't bound by pre-set dimensions. John Allsopp's *A Dao Of Web Design* called on practitioners to acknowledge this:

> *The control which designers know in the print medium, and often desire in the web medium, is simply a function of the limitation of the printed page. We should embrace the fact that the web doesn't have the same constraints, and design for this flexibility.*

This call to arms went unheeded. Designers remained in their Matrix-like consensual hallucination where everyone's browser was the same width. That's understandable. There's a great comfort to be had in believing a reassuring fiction, especially when it confers the illusion of control.

There is another reason why web designers clung to the comfort of their fixed-width layouts. The tools of the trade encouraged a paper-like approach to designing for the web.

Ship of tools

It's a poor craftsperson who always blames their tools. And yet every craftsperson is influenced by their choice of tools. As Marshall McLuhan's colleague John Culkin put it, "we shape our tools and thereafter our tools shape us."

When the discipline of web design was emerging, there was no software created specifically for visualising layouts on the web. Instead designers co-opted existing tools.

Adobe Photoshop was originally intended for image manipulation; touching up photos, applying filters, compositing layers, and so on. By the mid nineties it had become an indispensable tool for graphic designers. When those same designers began designing for the web, they continued using the software they were already familiar with.

If you've ever used Photoshop then you'll know what happens when you select "New" from the "File" menu: you will be asked to enter fixed dimensions for the canvas you are about to work within. Before adding a single pixel, a fundamental design decision has been made that reinforces the consensual hallucination of an inflexible web.

Photoshop alone can't take the blame for fixed-width thinking. After all, it was never intended for designing web pages. Eventually, software was released with the specific goal of creating web pages. Macromedia's Dreamweaver was an early example of a web design tool. Unfortunately it operated according to the idea of WYSIWYG: What You See Is What You Get.

While it's true that when designing with Dreamweaver, what *you* see is what *you* get, on the web there is no guarantee that what you see is what everyone else will get. Once again, web designers were encouraged to embrace the illusion of control rather than face the inherent uncertainty of their medium.

It's possible to overcome the built-in biases of tools like Photoshop and Dreamweaver, but it isn't easy. We might like to think that we are in control of our tools, that we bend them to our will, but the truth is that all software is opinionated software. As futurist Jamais Cascio put it, "software, like all technologies, is inherently political":

> *Code inevitably reflects the choices, biases and desires of its creators.*

Small wonder then that designers working with the grain of their tools produced websites that mirrored the assumptions baked into those tools—assumptions around the ability to control and tame the known unknowns of the World Wide Web.

Reality bites

By the middle of the first decade of the twenty-first century, the field of web design was propped up by multiple assumptions:

- that everyone was browsing with a screen large enough to view a 960 pixel wide layout;
- that everyone had broadband internet access, mitigating the need to optimise the number and file size of images on web pages;
- that everyone was using a modern web browser with the latest plug-ins installed.

A minority of web designers were still pleading for fluid layouts. I counted myself amongst their number. We were tolerated in much the same manner as a prophet of doom on the street corner wearing a sandwich board reading "The End Is Nigh"—an inconvenient but harmless distraction.

There were even designers suggesting that Photoshop might not be the best tool for the web, and that we could consider designing directly in the browser using CSS and HTML. That approach was criticised as being too constraining. As we've seen, Photoshop has its own constraints but those had been internalised by designers so comfortable in using the tool that they no longer recognised its shortcomings.

This debate around the merits of designing Photoshop comps and designing in the browser would have remained largely academic if it weren't for an event that would shake up the world of web design forever.

Stuck inside of mobile

> *An iPod. A phone. And an internet communicator. An iPod. A phone …are you getting it? These are not three separate devices. This is one device. And we are calling it: iPhone.*

With those words in 2007, Steve Jobs unveiled a mobile device that could be used to browse the World Wide Web.

$599
No Service
4:14 PM
Adactio: Journal
http://adactio.com/journal/
adactio
Friday, August 24th, 2007
Illinois apart
Things are busy, busy, busy at Clearleft. As well as the regular client work, we're on the home stretch for dConstruct 2007. I've been slacking off with the podcast and I really need to knuckle down and hammer out the format for the upcoming microformats workshop. The food blog and DOM Scripting blog have been sorely neglected too but that's nothing new.
I really shouldn't be taking any time out right now but I can't resist the lure of a few days in Chicago. I have the great honour of being asked to speak at An Event Apart, the traveling roadshow for people who make websites. Tomorrow morning I'll be getting the bus to Heathrow where a plane awaits, ready to whisk me across the pond.
I've never been to Chicago before. I hear the skyline is breathtaking. I've been reading up on the local attractions on offer and before the conference kicks off I'm hoping to make it to the aquarium and the Field Museum (where a Darwin exhibition is currently running).
Most of all, I'm determined to sample the famous pizza and hot dogs. My explorations will, of course, be documented on Flickr.
3

The iPhone.

Web-capable mobile devices existed before the iPhone, but they were mostly limited to displaying a specialised mobile-friendly file format called WML. Very few devices could render HTML. With the introduction of the iPhone and its competitors, handheld devices were shipping with modern web browsers capable of being first-class citizens on the web. This threw the field of web design into turmoil.

Assumptions that had formed the basis for an entire industry were now being called into question:

- How do we know if people are using wide desktop screens or narrow handheld screens?
- How do we know if people are browsing with a fast broadband connection at home or with a slow mobile network?
- How do we know if a device even supports a particular technology or plug-in?

The rise of mobile devices was confronting web designers with the true nature of the web as a flexible medium filled with unknowns.

The initial reaction to this newly-exposed reality involved segmentation. Rather than rethink the existing desktop-optimised website, what if mobile devices could be shunted off to a separate silo? This mobile ghetto was often at a separate subdomain to the "real" site: m.example.com or mobile.example.com.

This segmented approach was bolstered by the use of the term "the mobile web" instead of the more accurate term "the web as experienced on mobile." In the tradition of their earlier consensual hallucinations, web designers were thinking of mobile and desktop not just as separate classes of device, but as entirely separate websites.

Determining which devices were sent to which subdomain required checking the browser's user-agent string against an ever-expanding list of known browsers. It was a Red Queen's race just to stay up to date. As well as being error-prone, it was also fairly arbitrary. While it might have once been easy to classify, say, an iPhone as a mobile device, that distinction grew more difficult over time. With the introduction of tablets such as the iPad, it was no longer clear which devices should be redirected to the mobile URL. Perhaps a new subdomain was called for—t.example.com or tablet.example.com—along with a new term like "the tablet web". But what about the "TV web" or the "internet-enabled fridge web?"

We are one

The practice of creating different sites for different devices just didn't scale. It also ran counter to a long-held ideal called One Web:

> *One Web means making, as far as is reasonable, the same information and services available to users irrespective of the device they are using.*

But this doesn't mean that small-screen devices should be served page layouts that were designed for larger dimensions:

> *However, it does not mean that exactly the same information is available in exactly the same representation across all devices.*

If web designers wished to remain true to the spirit of One Web, they needed to provide the same core content at the same URL to everyone regardless of their device. At the same time, they needed to be able to create different layouts depending on the screen real-estate available.

The shared illusion of a one-size-fits-all approach to web design began to evaporate. It was gradually replaced by an acceptance of the ever-changing fluid nature of the web.

Positive response

In April of 2010 Ethan Marcotte stood on stage at An Event Apart in Seattle, a gathering for people who make websites. He spoke about an interesting school of thought in the world of architecture: responsive design, the idea that buildings could change and adapt according to the needs of the people using the building. This, he explained, could be a way to approach making websites.

One month later he expanded on this idea in an article called *Responsive Web Design*. It was published on A List Apart, the same website that had published John Allsopp's *A Dao Of Web Design* ten years earlier. Ethan's article shared the same spirit as John's earlier rallying cry. In fact, Ethan begins his article by referencing *A Dao Of Web Design*.

Both articles called on web designers to embrace the idea of One Web. But whereas *A Dao Of Web Design* was largely rejected by designers comfortable with their WYSIWYG tools, *Responsive Web Design* found an audience of designers desperate to resolve the mobile conundrum.

The adjacent possible

Writer Steven Johnson has documented the history of invention and innovation. In his book *Where Good Ideas Come From*, he explores an idea called "the adjacent possible":

> *At every moment in the timeline of an expanding biosphere, there are doors that cannot be unlocked yet. In human culture, we like to think of breakthrough ideas as sudden accelerations on the timeline, where a genius jumps ahead fifty years and invents something that normal minds, trapped in the present moment, couldn't possibly have come up with. But the truth is that technological (and scientific) advances rarely break out of the adjacent possible; the history of cultural progress is, almost without exception, a story of one door leading to another door, exploring the palace one room at a time.*

This is why the microwave oven could not have been invented in medieval France; there are too many preceding steps required—manufacturing, energy, theory—to make that kind of leap. Facebook could not exist without the World Wide Web, which could not exist without the internet, which could not exist without computers, and so on. Each step depends upon the accumulated layers below.

By the time Ethan coined the term Responsive Web Design a number of technological advances had fallen into place. As I wrote in the foreword to Ethan's subsequent book on the topic:

> *The technologies existed already: fluid grids, flexible images, and media queries. But Ethan united these techniques under a single banner, and in so doing changed the way we think about web design.*

1. Fluid grids. The option to use percentages instead of pixels has been with us since the days of TABLE layouts.

2. Flexible images. Research carried out by Richard Rutter showed that browsers were becoming increasingly adept at resizing images. The intrinsic dimensions of an image need not be a limiting factor.

3. Media queries. Thanks to the error-handling model of CSS, browsers had been adding feature upon feature over time. One of those features was CSS media queries—the ability to define styles according to certain parameters, such as the dimensions of the browser window.

The layers were in place. A desire for change—driven by the relentless rise of mobile—was also in place. What was needed was a slogan under which these could be united. That's what Ethan gave us with Responsive Web Design.

Changing mindset

The first experiments in responsive design involved retrofitting existing desktop-centric websites: converting pixels to percentages, and adding media queries to remove the grid layout on smaller screens. But this reactive approach didn't provide a firm foundation to build upon. Fortunately another slogan was able to encapsulate a more resilient approach.

Luke Wroblewski coined the term Mobile First in response to the ascendency of mobile devices:

> *Losing 80% of your screen space forces you to focus. You need to make sure that what stays on the screen is the most important set of features for your customers and your business. There simply isn't room for any interface debris or content of questionable value. You need to know what matters most.*

If you can prioritise your content and make it work within the confined space of a small screen, then you will have created a robust, resilient design that you can build upon for larger screen sizes.

Stephanie and Bryan Rieger encapsulated the mobile-first responsive design approach:

> *The lack of a media query is your first media query.*

In this context, Mobile First is less about mobile devices per se, and instead focuses on prioritising content and tasks regardless of the device. It discourages assumptions. In the past, web designers had fallen foul of unfounded assumptions about desktop devices. Now it was equally important to avoid making assumptions about mobile devices.

Web designers could no longer make assumptions about screen sizes, bandwidth, or browser capabilities. They were left with the one aspect of the website that was genuinely under their control: the content.

Echoing *A Dao Of Web Design*, designer Mark Boulton put this new approach into a historical context:

> *Embrace the fluidity of the web. Design layouts and systems that can cope to whatever environment they may find themselves in. But the only way we can do any of this is to shed ways of thinking that have been shackles around our necks. They're holding us back. Start designing from the content out, rather than the canvas in.*

This content-out way of thinking is fundamentally different to the canvas-in approach that dates all the way back to the Book of Kells. It asks web designers to give up the illusion of control and create a materially-honest discipline for the World Wide Web.

Relinquishing control does not mean relinquishing quality. Quite the opposite. In acknowledging the many unknowns involved in designing for the web, designers can craft in a resilient flexible way that is true to the medium.

Texan web designer Trent Walton was initially wary of responsive design, but soon realised that it was a more honest, authentic approach than creating fixed-width Photoshop mock-ups:

> *My love for responsive centers around the idea that my website will meet you wherever you are—from mobile to full-blown desktop and anywhere in between.*

For years, web design was dictated by the designer. The user had no choice but to accommodate the site's demand for a screen of a certain size or a network connection of a certain speed. Now, web design can be a conversation between the designer and the user. Now, web design can reflect the underlying principles of the web itself.

On the twentieth anniversary of the World Wide Web, Tim Berners-Lee wrote an article for Scientific American in which he reiterated those underlying principles:

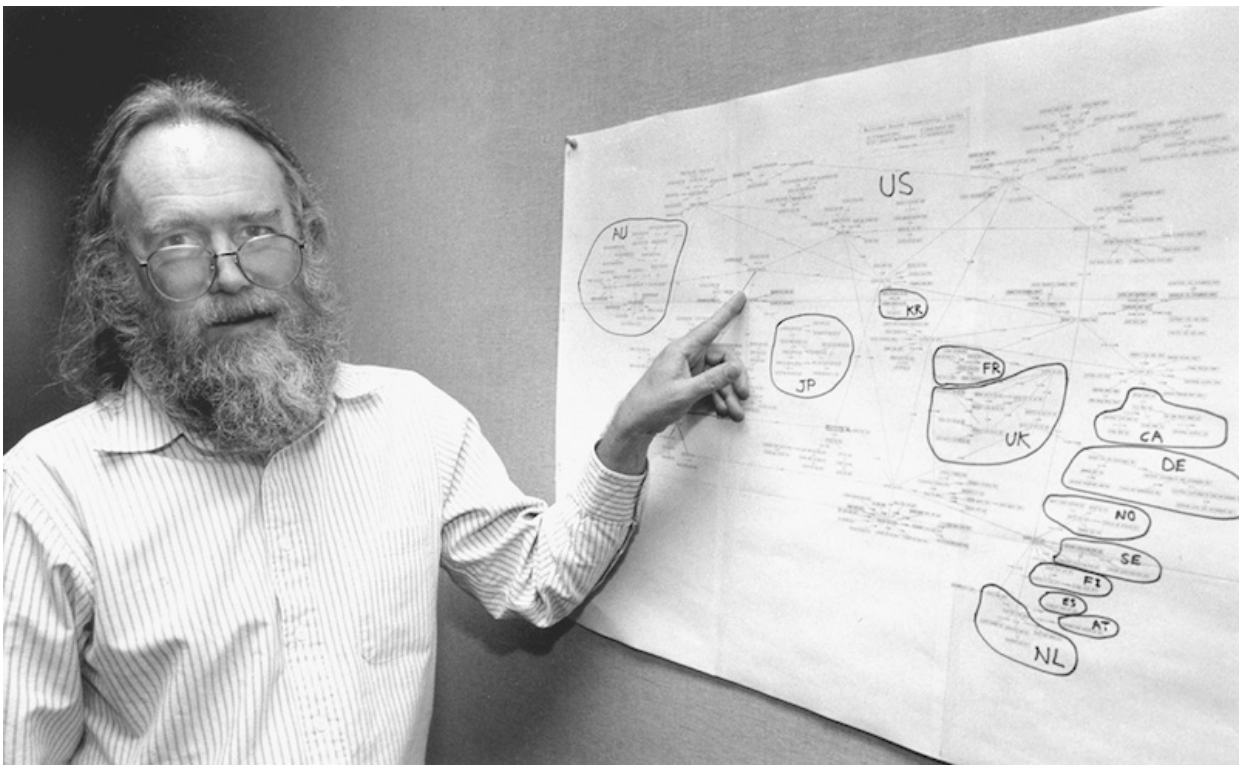
> *The primary design principle underlying the Web's usefulness and growth is universality. The Web should be usable by people with disabilities. It must work with any form of information, be it a document or a point of data, and information of any quality—from a silly tweet to a scholarly paper. And it should be accessible from any kind of hardware that can connect to the Internet: stationary or mobile, small screen or large.*

RESILIENT WEB DESIGN

CHAPTER 4:

Languages

Jon Postel was one of the engineers working on the ARPANET, the precursor to the internet. He wanted to make sure that the packets—or "datagrams"—being shuttled around the network were delivered in the most efficient way. He came to realise that a lax approach to errors was crucial to effective packet switching.



Jon Postel.
Photograph by Irene Fertik, USC News Service.
Copyright 1994, USC.

If a node on the network receives a datagram that has errors, but is still understandable, then the packet should be processed anyway. Conversely every node on the network should attempt to send well-formed packets. This line of thinking was enshrined in the Robustness Principle, also known as Postel's Law:

> *Be conservative in what you send; be liberal in what you accept.*

If that sounds familiar, it's because that's the way that web browsers deal with HTML and CSS. Even if there are errors in the HTML or CSS, the browser will still attempt to process the information, skipping over any pieces that it can't parse.

Declaration

HTML and CSS are both examples of declarative languages. An author writing in a declarative language describes a desired outcome without providing step-by-step instructions to the computer processing the file. With HTML, you can describe the nature of the content—paragraphs, headings, form fields, etc.—without having to explain exactly what the browser should do with that information. With CSS, you can describe the desired appearance of the content—colours, borders, etc.—without having to write a program to apply those styles.

Most programming languages are not declarative, they are imperative. Perl, Java, C++ …these are all examples of imperative languages. If you're writing in one of those languages, you must provide precise instructions to the computer interpreting your code.

Imperative languages provide you with more power and precision than declarative languages. That comes at a price. Imperative languages tend to be harder to learn than declarative languages. It's also harder to apply Postel's Law to imperative languages. If you make a single mistake—one misplaced comma or semi-colon—the entire program may fail. A misspelt tag in HTML or a missing curly brace in CSS can also cause headaches, but imperative programs must be well-formed or they won't run at all.

Imperative languages such as PHP, Ruby, and Python can be found on the servers powering the World Wide Web, reading and writing database records, processing input, and running complex algorithms. You can choose just about any programming language you want when writing server-side code. Unlike the unknowability of the end user's web browser, you have control over your server's capabilities.

If you want to write imperative code that runs in a web browser, you only have one choice: JavaScript.

Scripting

The idea of executing a program from within a web page is as old as the web itself. Here's an early email to the www-talk mailing list:

> *I would like to know, whether anybody has extended www such, that it is possible to start arbitrary programs by hitting a button in a www browser.*

Tim Berners-Lee, creator of the World Wide Web, responded:

> *Very good question. The problem is that of programming language. You need something really powerful, but at the same time ubiquitous. Remember a facet of the web is universal readership. There is no universal interpreted programming language.*

That was in 1992. The universal interpreted programming language finally arrived in 1996. It was written in ten days by a programmer at Netscape named Brendan Eich.

The language went through a few name changes. First it was called Mocha. Then it was officially launched as LiveScript. Then the marketing department swept in and renamed it JavaScript, hoping that the name would ride the wave of hype associated with the then-new Java language. In truth, the languages have little in common. Java is to JavaScript as ham is to hamster.

Patterns of progress

JavaScript gave designers the power to update web pages even after they had loaded. Two common uses soon emerged: rollovers and form validation.

Swapping out images when someone hovers their cursor over a link might not seem like a sensible use of a brand new programming language, but back in the nineties there was no other way of creating hover effects.

Before JavaScript came along, a form would have to be submitted to a web server before you could check to make sure that all the required fields were filled in, or that the information that was entered corresponded to an expected format.

Both of those use cases still exist, but now there's no need to reach for JavaScript. You can create rollover effects using the `:hover` pseudo-class in CSS. You can validate form fields using the REQUIRED and TYPE attributes in HTML.

That's a pattern that repeats again and again: a solution is created in an imperative language and if it's popular enough, it migrates to a declarative language over time. When a feature is available in a declarative language, not only is it easier to write, it's also more robust.

The loose error-handling of HTML and CSS means that many authoring mistakes or browser support issues are handled gracefully; the browser simply ignores what it doesn't understand and carries on. That's often good enough. By contrast, if you give a browser some badly-formed JavaScript or attempt to use an unsupported JavaScript feature, not only will the browser throw an error, it will stop parsing the script at that point and refuse to go any further.

Responsibility

JavaScript gave web designers the power to create websites that were slicker, smoother, and more reactive. The same technology also gave web designers the power to create websites that were sluggish, unwieldy, and more difficult to use.

One of the earliest abuses of JavaScript came (unsurprisingly) from the advertising industry, a business whose very raison d'être is often at odds with the goals of people trying to achieve a task as quickly as possible. JavaScript allows you to create new browser windows, something that previously could only be done by the user. A young developer named Ethan Zuckerman realised that he could spawn a new window with an advertisement in it. That allowed advertisers to put their message in front of website visitors. Not only that, but JavaScript could be used to spawn multiple windows, some of them visible, some of them hidden behind the current window. It was a fiendish solution.

Twenty years later, Zuckerman wrote:

> *I wrote the code to launch the window and run an ad in it. I'm sorry.*

Pop-up (and pop-under) windows became so intolerable that browsers had to provide people with a means to block them.

The advertising industry later found other ways to abuse JavaScript. Ad-supported online publishers injected bloated and inefficient JavaScript into their pages, making them slow to load. JavaScript was also used to track people from site to site. People reached for ad-blocking software to combat this treatment. Eventually ad blocking was built into browsers and operating systems to give us the ability to battle excessive JavaScript.

Web designers would do well to remember what the advertising industry chose to ignore: on the web, the user has the final say.

2.0

The rise of JavaScript was boosted in 2005 with the publication of an article entitled *Ajax: A New Approach to Web Applications* by Jesse James Garrett. The article put a name to a technique that was gaining popularity. Using a specific subset of JavaScript, it was possible for a web browser to send and receive data from a web server without refreshing the whole page. The result was a smoother user experience.

The term Ajax was coined at the same time that another neologism was in the ascendent. Tim O'Reilly used the phrase Web 2.0 to describe a new wave of web products and services. Unlike Ajax, it was difficult to pin down a definition of Web 2.0. For business people, it meant new business models. For graphic designers, it meant rounded corners and gradients. For developers, it meant JavaScript and Ajax.

Whatever its exact meaning, the term Web 2.0 captured a mood and a feeling. Everything was going to be different now. The old ways of thinking about building for the web could be cast aside. Treating the web as a limitless collection of hyperlinked documents was passé. The age of web apps was at hand.

Appiness

In the 1964 supreme court case Jacobellis versus Ohio, Justice Potter Stewart provided this definition of obscenity:

> *I know it when I see it.*

The same could be said for Web 2.0, or for the term "web app." We can all point to examples of web apps, but it's trickier to provide a definition for the term. Web apps allow people to create, edit, and delete content. But these tasks were common long before web apps arrived. People could fill in forms and submit them to a web server for processing. Ajax removed the necessity for that round trip to the server.

Perhaps the definition of a web app requires some circular reasoning:

- JavaScript is a requirement for a web app, and
- a web app is a website that requires JavaScript to work.

In that case, building web apps depends on a fundamental assumption: JavaScript must be available and reliable. But because of its imperative nature, JavaScript tends to be more fragile than a declarative language like HTML. Relying on JavaScript might not be such a safe assumption after all.

Unforgiven

HTML's loose error-handling allowed it to grow in power over time. It also ensured that the language was easy to learn. Even if you made a mistake, the browser's implementation of Postel's Law ensured that you'd still get a result. Surprisingly, there was an attempt to remove this superpower from HTML.

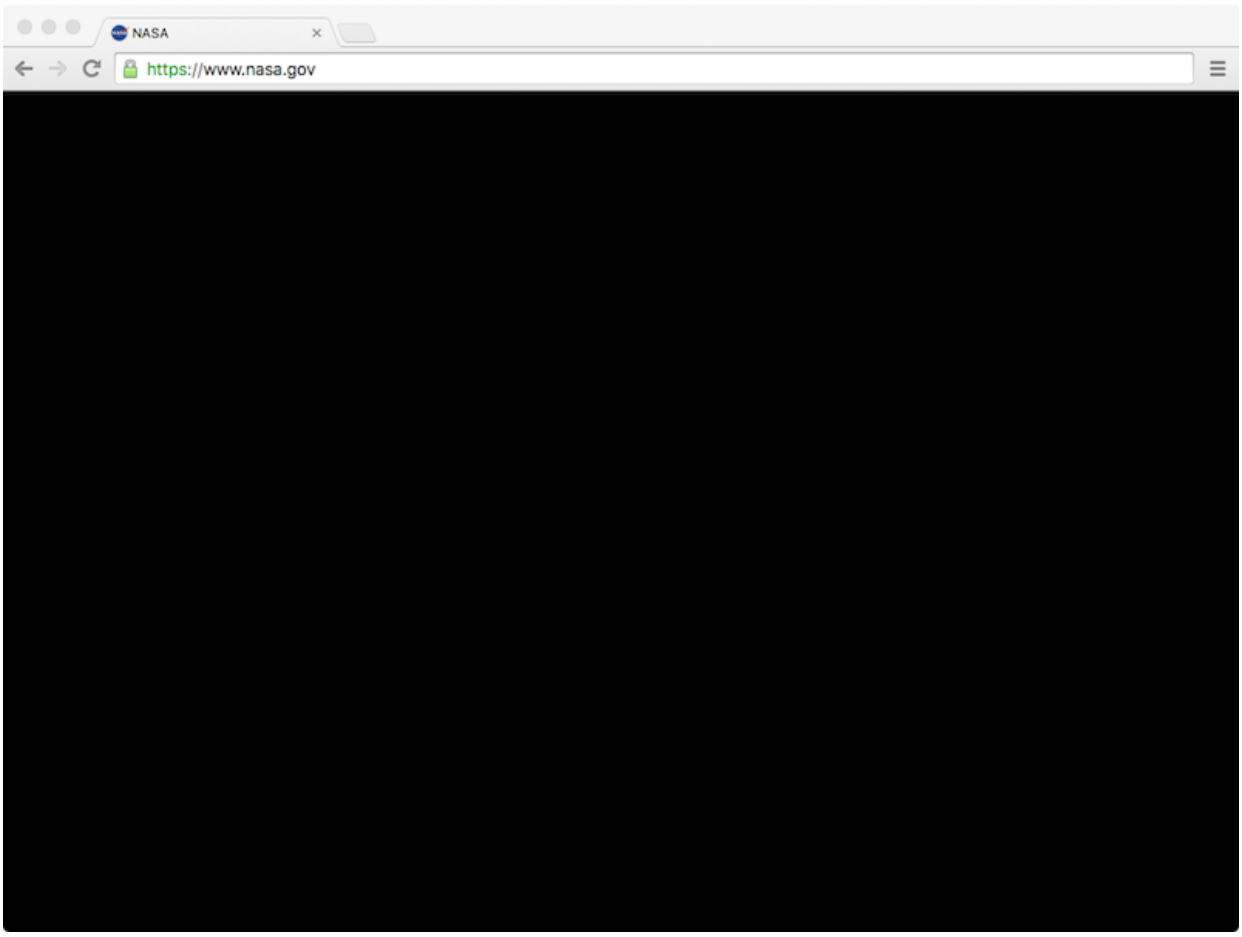
After the standardisation of HTML version 4 in 1999, the World Wide Web Consortium published XHTML 1.0. This reformulated HTML according to the rules of the XML data format. Whereas HTML can have uppercase or lowercase tag names and attributes, XML requires them to be all lowercase. There were some other differences: all attributes had to be quoted, and standalone elements like IMG or BR required a closing slash.

XHTML 1.0 didn't add any new features to the language. It was simply a stricter way of writing markup. XHTML 2.0 was a different proposition. Not only would it remove established elements like IMG, it would also implement XML's draconian error-handling model. If there is a single error in an XML document—one unquoted attribute or missing closing slash—then the parser should stop immediately and refuse to render anything.

XHTML 2.0 died on the vine. Its theoretical purity was roundly rejected by the people who actually made websites for a living. Web designers rightly refused to publish in a format that would fail completely instead of trying to recover from an error.

Strange then, that years later, web designers would happily create entire websites using JavaScript, a language that shares XML's unforgiving error-handling model. They didn't call them websites. They called them web apps. That distinction was cold comfort to someone who couldn't complete their task because a service relied on JavaScript for crucial functionality.

Despite JavaScript's fragile error-handling model, web designers became more reliant on JavaScript over time. In 2015, NASA relaunched its website as a web app. If you wanted to read the latest news of the agency's space exploration efforts, you first had to download and execute three megabytes of JavaScript. This content—text and images—could have been delivered in the HTML, but the developers decided to use Ajax to retrieve this data instead. Until all that JavaScript was loaded, parsed, and executed, visitors to the site were left staring at a black background. Perhaps this was intended as a demonstration of the vast lonely emptiness of space.



The 2015 version of the nasa.gov website with incomplete JavaScript.

This highlights another difference between HTML and JavaScript. Whereas HTML can be rendered piece by piece as it is downloaded, a JavaScript file must be downloaded in its entirety before its contents can be parsed. While it's tempting to think that only a small minority of visitors will miss out on a site's JavaScript, the truth is that everybody is a non-JavaScript user until the JavaScript has finished loading ...*if* the JavaScript finishes loading. Flaky connections, interfering network operators, and unpredictable ad-blocking software can torpedo the assumption that JavaScript will always be available.

The problem is not with people deliberately disabling JavaScript in their browsers. Although that's a use case worth considering, it's not the most common cause of JavaScript errors. Stuart Langridge put together a list of all the potential points of failure under the title *Everyone has JavaScript, right?*

> *The user requests your web app. Has the page loaded yet? Did the HTTP request for the JavaScript succeed? Did the HTTP request for the JavaScript complete? Does the corporate firewall block JavaScript? Does their ISP or mobile operator interfere with downloaded JavaScript? Have they switched off JavaScript? Do they have add-ons or plug-ins installed which inject script or alter the DOM in ways you didn't anticipate? Is the Content Delivery Network up? Does their browser support the JavaScript you've written?*

Many of those problems would also affect HTML and CSS files, but because of Postel's Law, they can recover gracefully.

This doesn't mean that web designers shouldn't use JavaScript. But it does mean that web designers shouldn't *rely* on JavaScript when a simpler solution exists.

Platform

Web designers who ignored the message of John Allsopp's *A Dao of Web Design* made the mistake of treating the web like print. The history of print has much to offer—hierarchy, typography, colour theory—but the web is a fundamentally different medium. The history of software development also has much to offer—architecture, testing, process—but again, the web remains its own medium.

It's tempting to apply the knowledge and learnings from another medium to the web. But it is more structurally honest to uncover the web's own unique strengths and weaknesses.

The language we use can subtly influence our thinking. In his book *Metaphors We Live By*, George Lakoff highlights the dangers of political language. Obvious examples are "friendly fire" and "collateral damage", but a more insidious example is "tax relief"—before a debate has even begun, taxation has been framed as something requiring relief.

On the face of it, the term "web platform" seems harmless. Describing the web as a platform puts it on par with other software environments. Flash was a platform. Android is a platform. iOS is a platform. But the web is not a platform. The whole point of the web is that it is *cross*-platform.

A platform provides a controlled runtime environment for software. As long as the user has that runtime environment, you can be confident that they will get exactly what you've designed. If you build an iOS app and someone has an iOS device, you know that they will get 100% of your software. But if you build an iOS app and someone has an Android device, they will get 0% of your software. You can't install an iOS app on an Android device. It's all or nothing.

The web isn't as binary as that. If you build something using web technologies, and someone visits with a web browser, you can't be sure how many of the web technologies will be supported. It probably won't be 100%. But it's also unlikely to be 0%. Some people will visit with iOS devices. Others will visit with Android devices. Some people will get 80% or 90% of what you've designed. Others will get just 20%, 30%, or 50%. The web isn't a platform. It's a continuum.

Thinking of the web as a platform is a category error. A platform like Flash, iOS, or Android provides stability and certainty, but only under a very specific set of circumstances—your software must be accessed with the right platform-specific runtime environment. The web provides no such certainty, but it also doesn't restrict the possible runtime environments.

Platforms are controlled and predictable. The World Wide Web is chaotic and unpredictable.

The web is a hot mess.

RESILIENT WEB DESIGN

CHAPTER 5:

Layers

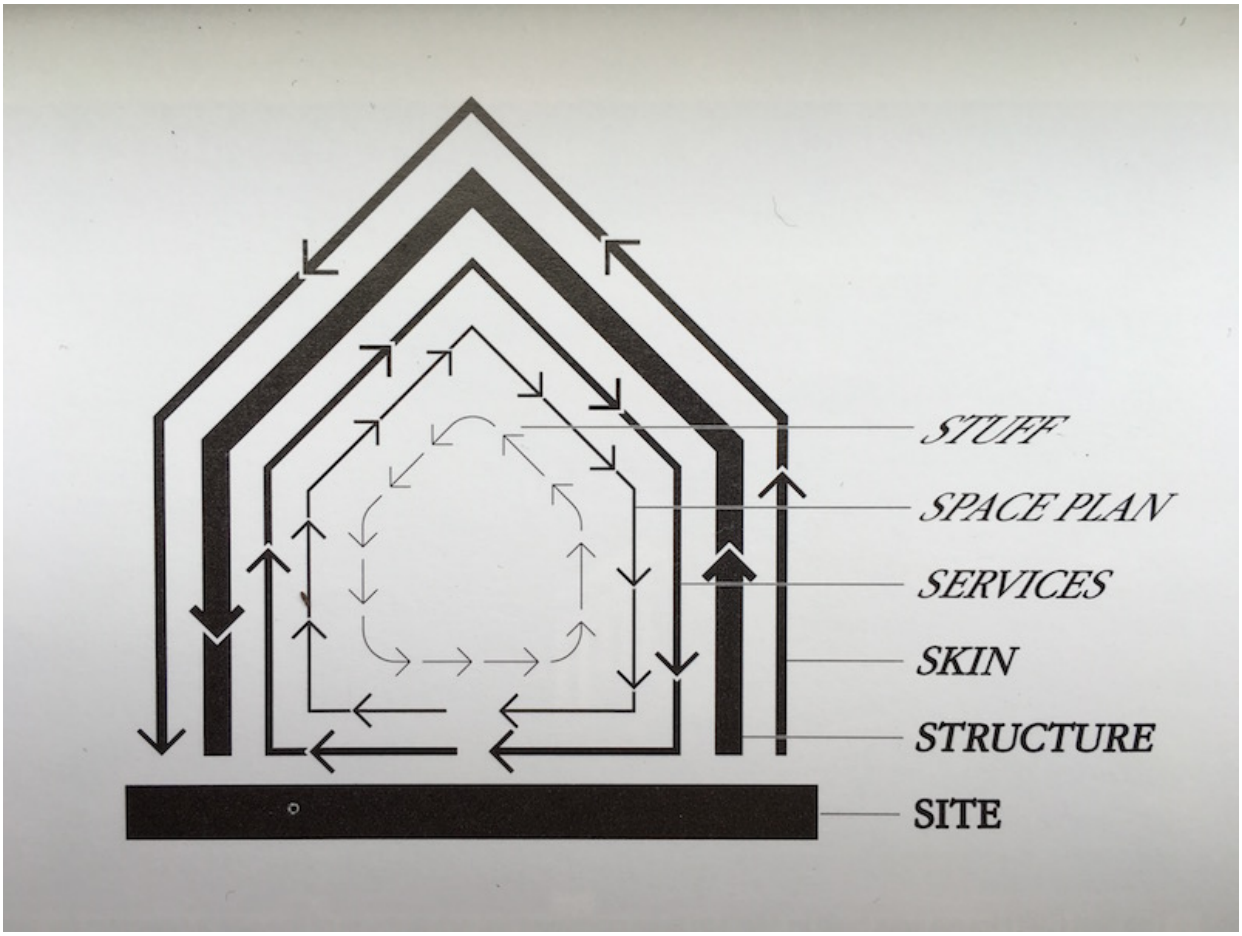
In his classic book

How Buildings Learn Stewart Brand highlights an idea by the British architect Frank Duffy:

> *A building properly conceived is several layers of longevity.*

Duffy called these shearing layers. Each of the layers moves at a different timescale. Brand expanded on the idea, proposing six alliterative layers:

1. **Site**—the physical location of a building only changes on a geological timescale.
2. **Structure**—the building itself can last for centuries.
3. **Skin**—the exterior surface gets a facelift or a new lick of paint every few decades.
4. **Services**—the plumbing and wiring need to be updated every ten years or so.
5. **Space plan**—the layout of walls and doors might change occasionally.
6. **Stuff**—the arrangement of furniture in a room can change on a daily basis.



Shearing layers.

The idea of shearing layers can also be applied to our creations on the web. Our domain names are the geological sites upon which we build. At the other end of the timescale, content on the web—the "stuff"—can be added and updated by the hour, the minute, or even the second. In between are the layers of structure, presentation, and behaviour: HTML, CSS, and JavaScript.

Those layers can be loosely-coupled, but they aren't completely independent. Just as a building cannot have furniture without first having rooms and walls, a style sheet needs some markup to act upon. The coupling between structure and presentation is handled through selectors in CSS: element selectors, class selectors, and so on. With JavaScript, the coupling is handled through the vocabulary of the Document Object Model, or DOM.

In a later book, *The Clock Of The Long Now*, Stewart Brand applied the idea of shearing layers—or pace layers—to civilisation itself. The slowest moving layer is nature, then there's culture, followed by governance, then infrastructure, and finally commerce and fashion are the fastest layers. In a loosely-coupled way, each layer depends on the layer below. In turn, the accumulation of each successive layer enables an "adjacent possible" filled with more opportunities.

Likewise, the expressiveness of CSS and JavaScript is only made possible on a foundation of HTML, which itself requires a URL to be reachable, which in turn depends on the HyperText Transfer Protocol, which sits atop the bedrock of TCP/IP.

Each of the web's shearing layers can be peeled back to reveal a layer below. Running that process in reverse—applying each layer in turn—is a key principle of resilient web design.

Progressive enhancement

In 2003, the South by Southwest festival in Austin, Texas was primarily an event for musicians and filmmakers. Today the music and film portions are eclipsed by the juggernaut of South by Southwest Interactive, dedicated to all things digital. In 2003, South by Southwest Interactive was a small affair, squeezed into one corner of one floor of the Austin Convention Center. It was a chance for a few web designers and bloggers to get together and share ideas. That year, Steven Champeon and Nick Finck presented a talk entitled *Inclusive Web Design For the Future with Progressive Enhancement*. They opened with this call to arms:

> *Web design must mature and accept the developments of the past several years, abandon the exclusionary attitudes formed in the rough and tumble dotcom era, realize the coming future of a wide variety of devices and platforms, and separate semantic markup from presentation logic and behavior.*

Like Tim Berners-Lee, Steven Champeon had experience of working with SGML, the markup language that would so heavily influence HTML. In dealing with documents that needed to be repurposed for different outputs, he came to value the separation of structure from presentation. A meaningfully marked up document can be presented in multiple ways through the addition of CSS and JavaScript.

This layered approach to the web allows the same content to be served up to a wide variety of people. But this doesn't mean that everyone gets the same experience. Champeon realised that a strong separation of concerns would allow enhancements to be applied according to the capabilities of the end user's device.

To paraphrase Karl Marx, progressive enhancement allows designers to ask from each browser according to its ability, and to deliver to each device according to its needs.

Do websites need to look exactly the same in every browser?

Some web designers were concerned that progressive enhancement would be a creative straitjacket. Designing for the lowest common denominator did not sound like a recipe for progress. But this was a misunderstanding. Progressive enhancement asks that designers *start* from the lowest common denominator (a well marked-up document), but there is no limit to where they can go from there.

In fact, it's the very presence of a solid baseline of HTML that allows web designers to experiment with the latest and greatest CSS. Thanks to Postel's Law and the loose error-handling model of CSS, designers are free to apply styles that only work in the newest browsers.

This means that not everyone will experience the same visual design. This is not a bug. This is a feature of the web. New browsers and old browsers; monochrome displays and multi-coloured displays; fast connections and slow connections; big screens, small screens, and no screens; everyone can access your content. That content *should* look different in such varied situations. If a website looks the same on a ten-year old browser as it does in the newest devices, then it probably isn't taking advantage of the great flexibility that the web offers.

To emphasis this, designer Dan Cederholm created a website to answer the question, "Do websites need to look exactly the same in every browser?" You can find the answer to that question at the URL:

dowebsitesneedtolookexactlythesameineverybrowser.com

At the risk of spoiling the surprise for you, the answer is a resounding "No!" If you visit that website, you will see that answer proudly displayed. But depending on the capabilities of your browser, you may or may not see some of the stylistic flourishes applied to that single-word answer. Even if you don't get *any* of the styles, you'll still get the content marked up with semantic HTML.

Cutting the mustard

Separating structure and presentation is relatively straightforward. You can declare whatever styles you want, safe in the knowledge that browsers will ignore what they don't understand. Separating structure and behaviour isn't quite so easy. If you give a browser some JavaScript that it doesn't understand, not only will it not apply the desired behaviour, it will refuse to parse the rest of the JavaScript.

Before you use a particular feature in JavaScript, it's worth testing to see if that feature is supported. This kind of feature detection can save your website's visitors from having a broken experience because of an unsupported feature. If you want to use Ajax, check first that the browser supports the object you're about to use to enable that Ajax functionality. If you want to use the geolocation API, check first that the browser supports it.

A team of web developers working on the BBC news website referred to this kind of feature detection as cutting the mustard. Browsers that cut the mustard get an enhanced experience. Browsers that don't cut the mustard still get access to the content, but without the JavaScript enhancements.

Feature detection, cutting the mustard, whatever you want to call it, is a fairly straightforward technique. Let's say you want to traverse the DOM using `querySelector` and attach events to some nodes in the document using `addEventListener`. Your mustard-cutting logic might look something like this:

```
if (document.querySelector &&
window.addEventListener) {
    // Enhance!
}
```

There are two points to note here:

1. This is feature detection, not browser detection. Instead of asking "which browser are you?" and trying to infer feature support from the answer, it is safer to simply ask "do you support this feature?"

2. There is no `else` statement.

Aggressive enhancement

Back when web designers were trying to exert print-like control over web browsers, a successful design was measured in pixel perfection: did the website look exactly the same in every browser?

Unless every browser supported a particular feature—like, say, rounded corners in CSS—then that feature was off the table. Instead, designers would fake it with extra markup and images. The resulting websites lacked structural honesty. Not only was this a waste of talent and energy on the part of the designers, it was a waste of the capabilities of modern web browsers.

The rise of mobiles, tablets, and responsive design helped to loosen this restrictive mindset. It is no longer realistic to expect pixel-perfect parity across devices and browsers. But you'll still find web designers bemoaning the fact that they have to support an older outdated browser because a portion of their audience are still using it.

They're absolutely right. Anyone using that older browser should have access to the same content as someone using the latest and greatest web browser. But that doesn't mean they should get the same experience. As Brad Frost puts it:

> *There is a difference between support and optimization.*

Support every browser ...but *optimise* for none.

Some designers have misunderstood progressive enhancement to mean that *all* functionality must be provided to *everyone*. It's the opposite. Progressive enhancement means providing *core* functionality to everyone. After that, it's every browser for itself. Far from restricting what features you can use, progressive enhancement provides web designers with a way to safely use the latest and greatest features without worrying about older browsers. Scott Jehl of the agency Filament Group puts it succinctly:

> *Progressive Enhancement frees us to focus on the costs of building features for modern browsers, without worrying much about leaving anyone out. With a strongly qualified codebase, older browser support comes nearly for free.*

If a website is built using progressive enhancement then it's okay if a particular feature isn't supported or fails to load: Ajax, geolocation, whatever. As long as the core functionality is still available, web designers don't need to bend over backwards trying to crowbar support for newer features into older browsers.

You also get a website that's more resilient to JavaScript's error-handling model. Mat Marquis worked alongside Scott Jehl on the responsive website for the Boston Globe.
He noted:

> *Lots of cool features on the Boston Globe don't work when JS breaks; "reading the news" is not one of them.*

The trick is identifying what it is considered core functionality and what is considered an enhancement.

RESILIENT WEB DESIGN

CHAPTER 6:

Steps

"Always design a thing by considering it in its next larger context", said the Finnish architect Eliel Saarinen. "A chair in a room, a room in a house, a house in an environment, an environment in a city plan."

At first glance, web design appears to be a kind of graphic design. Using graphic design tools like Photoshop to design websites reinforces this view. But to get to the heart of a website's purpose, we should consider the interface in its larger context: what are people trying to accomplish?

When designing for the web, it's tempting to think in terms of interactions like swiping, tapping, clicking, scrolling, dragging and dropping. But very few people wake up in the morning looking forward to a day of scrolling and tapping. They're more likely to think in terms of reading, writing, sharing, buying and selling. Web designers need to see past the surface-level actions to find the more meaningful verbs beneath.

In their book *Designing With Progressive Enhancement*, the Filament Group describe a technique they call "the x-ray perspective":

> *Taking an X-ray perspective means looking "through" the complex widgets and visual styles of a design, identifying the core content and functional pieces that make up the page, and finding a simple HTML equivalent for each that will work universally.*

If you're not used to this approach to web design, it can take some getting used to. But after a while it becomes a habit and then it's hard *not* to examine interfaces in this way. It's like trying not to notice bad kerning, or trying not to see the arrow in the whitespace of the FedEx logo, or trying to not to remember that all ducks are actually wearing dog masks.

This duck's bill looks like the face of a dog.
Photograph by Asmaa Dee.
Licensed under a Creative Commons Attribution–NonCommercial–NoDerivs 2.0 Generic license.

Here's a three-step approach I take to web design:

1. Identify core functionality.
2. Make that functionality available using the simplest possible technology.
3. Enhance!

Identifying core functionality might sound like it's pretty straightforward, and after a bit of practice, it is. But it can be tricky at first to separate what's truly necessary from what's nice to have.

Information

Let's say you're a news provider. That right there is the core functionality—to provide news. There are many, many other services you could also provide; interactive puzzles, realtime notifications, and more. Valuable as those services are, they're probably not as important as making sure that people have access to news.

Scrutiny of Melania Trump's speech follows plagiarism allegations
NEWS
Turkey suspends 15,200 education staff
BREAKING NEWS
The New York Times
G.O.P. Off to Fiery Start at Convention
theguardian

A selection of news outlets.

With that core functionality identified, it's time to move on to step two: how can you make that core functionality available using the simplest possible technology?

Theoretically, a plain text file would be the simplest possible way of providing the news. But as we're talking specifically about the web, let's caveat this step: how can you make the core functionality available using the simplest possible **web** technology? That would be an HTML file served up at a URL.

Even at this early stage it's possible to overcomplicate things. The HTML could be unnecessarily bloated. The URL could be unnecessarily verbose; hard to share or recall.

Now that the news has been marked up with the appropriate HTML elements—articles, headings, paragraphs, lists, and images—it's time for step three: enhance!

By default, the news will be presented using the browser's own stylesheet. It's legible, but not exactly pleasurable. By applying your own CSS, you can sculpt the content into a more pleasing shape. Whitespace, leading, colour and contrast are all at your disposal. You can even use custom fonts—an enhancement that was impossible on the web for many years.

There's no guarantee that every browser will be capable of executing every CSS declaration that you throw at it. That's okay. Those browsers will ignore what they don't understand. Crucially, the news is still available to everyone, regardless of the CSS capabilities of their browser.

For browsers on large-screen devices, you can introduce layout. It might seem odd at first to think of layout as an enhancement, but that's the lesson of mobile-first responsive design. Consider the content first, then mark it up with a sensible source order, then apply layout declarations within media queries.

Thanks to ever-evolving nature of CSS, there are multiple ways of applying layout. Like Andy Tannenbaum said:

> *The nice thing about standards is that you have so many to choose from.*

Communication

Applying the three-step process to a news site is relatively straightforward. Catching up with the news is a fairly passive act. To really test this process, we need to apply it to something more interactive.

Suppose we were building a social network. The people using our tool need to be able to communicate with one another regardless of where in the world they are. The core functionality is sending and receiving messages.

Just some of the social networking sites that have graced the web.

Displaying messages in a web browser isn't difficult. There might be a lot of complexity on the server involving databases, syncing, queueing, and load balancing, but the HTML needed to structure a reverse-chronological list isn't very different from the HTML needed for a news site.

Sending a message from the browser to the web server requires HTML that is interactive. That's where forms come in. In this case, a form with a text input and a submit button should be enough, at least for the basic functionality.

People can now receive and respond to messages on our social network, no matter what kind of device or browser they are using. Now the trick is to improve the experience without breaking that fundamental activity.

If we were to leave the site in this HTML-only state, I don't think we'd be celebrating our company's IPO anytime soon. To really distinguish our service from the competition, we need that third step in the process: enhance!

At the very least, we can apply the same logic we used for the news site and style our service. Using CSS we can provide colour, texture, contrast, web fonts, and for larger screens, layout. But let's not stop with the presentation. Let's improve the interaction too.

Right now this social network has the same kind of page-based interaction as a news site. Every time someone sends a message to the server, the server sends back a whole new page to the browser. We can do better than that. Time for some Ajax.

We can intercept the form submission and send the data to the server using Ajax—I like using the word Hijax to describe this kind of Ajax interception. If there's a response from the server, we can also update part of the current page instead of refreshing the whole page. This would also be a good time to introduce some suitable animation.

We can go further. Browsers that support WebSockets can receive messages from the server. People using those browsers could get updates as soon as they've been sent. It's even possible to use peer-to-peer connections between browsers to allow people to communicate directly.

Not every browser supports this advanced functionality. That's okay. The core functionality—sending and receiving messages—is still available to everyone.

Creation

What if our social network were more specialised? Let's make it a photo-sharing service. That raises the bar a bit. Instead of sending and receiving messages, the core functionality is now sending and receiving images.

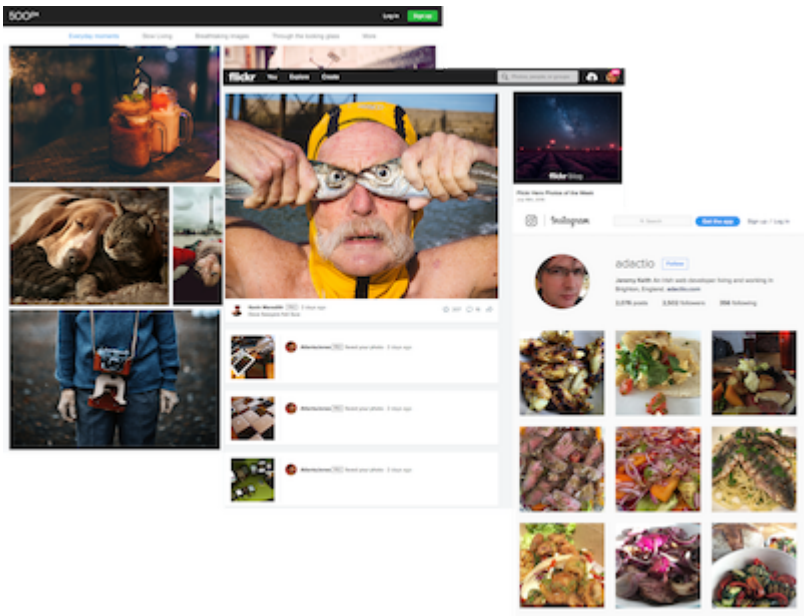


Photo-sharing websites.

The interface needs to show a reverse-chronological list of images. HTML can handle that. Once again we need a form to send data to the server, but this time it needs to be a file upload instead of a text field.

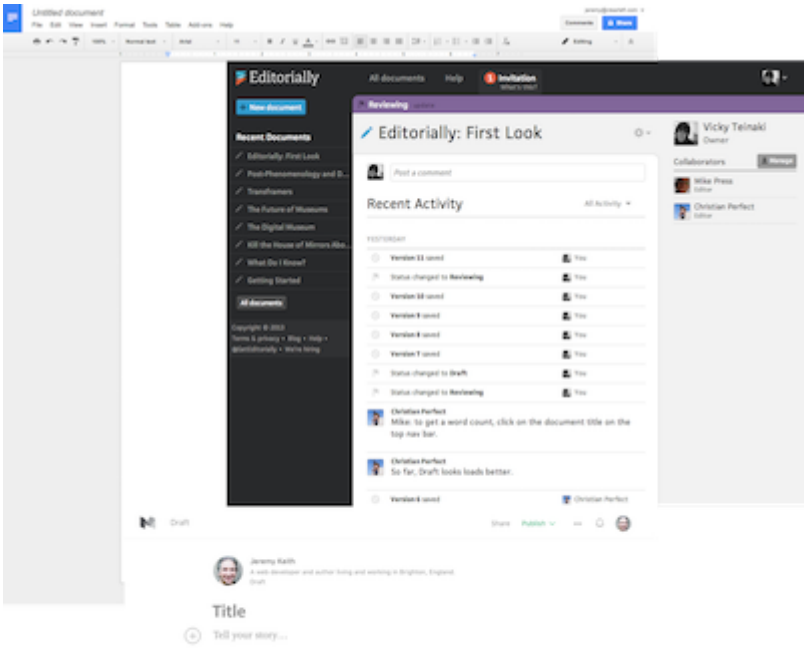
With those changes, the core functionality is in place. Time to enhance.

As well as all the existing enhancements—CSS, web fonts, Ajax, WebSockets—we could make use of the File API introduced in HTML5. This allows us to manipulate the image directly in the browser. We could apply effects to the image before sending it to the server. Using CSS filters, we can offer a range of image enhancements from sepia tones to vignettes. But if a browser doesn't support the File API or CSS filters, people can still upload their duck-facing selfies.

Collaboration

There was a time when using software meant installing separate programs on your computer. Today it's possible to have a machine with nothing more than a web browser installed on it. Writing emails, looking up contact details, making calendar appointments, bookkeeping and other financial tasks can all be done without having to install bespoke applications. Instead, the act of visiting a URL can conjure up the tool you need when you need it.

Delivering software over the web doesn't just replace the desktop-centric way of working. The presence of an internet connection opens up possibilities for all kinds of collaboration. Take, for example, the kinds of applications that were once called "word processors." As long as those programs were tethered to individual machines, trying to collaboratively edit a document was bound to be a tricky task requiring plenty of coordination, sending files back and forth. Using the web, the act of sharing a single URL could allow multiple people to work on the same document.



Collaborative writing tools on the web.

Let's apply the three-step process to a web-based word processor:

> *Identify core functionality.*

The tautological answer would be "processing words." Not very helpful. What do people actual *do* with this software? They write. They share. They edit.

> *Make that functionality available using the simplest possible technology.*

Looking at our three verbs—writing, sharing, and editing—we get one of them for free just by using URLs: sharing. The other two—writing and editing—require the use of a form. A basic TEXTAREA element can act as the receptacle for the words, sentences, and paragraphs that will make up everything from technical reports to the great American novel. Submitting that content to a web server means it can be saved for later.

Technically, that's a web-based word processor, accessible to anyone with a web browser and an internet connection. But the experience is clunky and dull. It would be a shame not to take advantage of some of slicker options available in modern browsers.

> *Enhance!*

Using JavaScript, the humble TEXTAREA can be replaced with a richer editing interface, detecting each keystroke and applying styling on the fly. Web fonts can make the writing experience more beautiful. Ajax will allow work to be saved to the server almost constantly, without the need for a form submission. WebSockets provide the means for multiple people to work on the same document at the same time.

Both Ajax and WebSockets require an internet connection to work. There's no guarantee of a stable internet connection, especially if you're trying to work on a train or in a hotel. Modern browsers provide features which, after the initial page load, can turn the network itself into an enhancement.

If a browser supports some form of local storage, then data can be stored in a client-side database. Flaky network connections or unexpected power outages won't get in the way of saving that important document. Using Service Workers, web developers can provide instructions on what to do when the browser (or the server) is offline.

These are modern browser features that we should be taking full advantage of …once we've made sure that we're providing a basic experience for everyone.

Scale

Ward Cunningham, the creator of the wiki, coined the term "technical debt" to describe a common problem in the world of software. Decisions made in haste at the beginning of a project lead to a cascade of issues further down the line. I like to think of the three-step layered approach as a kind of "technical credit." Taking the time to provide core functionality at the beginning gives you the freedom to go wild with experimentation from then on.

Some people have misunderstood progressive enhancement to mean foregoing the latest and greatest browser technologies, but in fact the opposite is true. Taking a layered approach to building on the web gives you permission to try cutting-edge JavaScript APIs, regardless of how many or how few browsers currently implement them.

We've looked at some examples of applying the three-step approach to a few products and services—news, social networking, photo sharing, and word processing. You can apply this approach to many more services: making and updating items in a to-do list, managing calendar appointments, looking up directions, making reservations at nearby restaurants. Each one can be built with the same process:

1. Identify core functionality.
2. Make that functionality available using the simplest possible technology.
3. Enhance!

This approach works at different scales. It doesn't just work at the highest level of the service; it can also be applied at the level of individual URLs within.

Ask "what is the core functionality of this URL?", make that functionality available using the simplest possible technology, and then enhance from there. This can really clarify which content is most important, something that's important in a mobile-first responsive workflow. Once you've established that, make sure that content is sent from the server as HTML (the simplest possible technology). Then, using conditional loading, you could decide to make Ajax requests for supporting content if the screen real-estate is available. For the URL of an individual news story, the story itself would be sent in the initial response, but related stories or comments could be pushed from the server only as needed (although you can still provide links to the related stories and comments for everyone).

We can go deeper. We can apply the three-step process at the scale of individual components within a page. "What is the core functionality of this component? How can I make that functionality available using the simplest possible technology? Now how can I enhance it?"

A component might be designed to be an all-singing all-dancing interactive map. With x-ray goggles, the core functionality reveals itself to be something much simpler: showing a location. Provide the address of that location in text: the simplest possible technology. Now you can enhance.

It's worth remembering that enhancements can be provided on a sliding scale. The first enhancement for a text address might be to provide a static image. The next level up from that would be to swap out the static image with an interactive Ajax-powered slippy map. If a browser supports the geolocation API, you could show the distance to the location. Layer on some animations and transitions to help convey the directions better.

Site navigation is another discrete component that lends itself well to a sliding scale of enhancements. The core functionality of navigation is to provide links to resources. The simplest—and still the best—technology to enable that is the humble hyperlink. A list of links should do the trick. With that in place, you are now free to enhance it into something really compelling. Off-canvas navigation, progressive disclosure, sliding, swiping, fading, expanding …the sky's the limit.

Because enhancements can be layered on according to the capabilities of each browser, it quickly becomes clear that this approach doesn't simply reduce down to having two versions of everything (the basic version and the enhanced version). Instead the service, the URLs, and the components you are designing could be experienced in any number of ways. And that's okay.

Websites do not need to look exactly the same in every browser.

RESILIENT WEB DESIGN

CHAPTER 7:

Challenges

THE FOURTH ANNUAL conference on hypertext took place in San Antonio, Texas in December 1991. Tim Berners-Lee's World Wide Web project was starting to take shape then. Thinking the conference organisers and attendees would appreciate the project, he submitted a proposal to Hypertext '91. The proposal was rejected.

The hypertext community felt that the World Wide Web project was far too simplistic. Almost every other hypertext system included the concept of two-way linking. If a resource moved, any links pointing to that resource would update automatically. The web provided no such guarantees. Its system of linking was much simpler—you just link to something and that's it. To the organisers of Hypertext '91, this seemed hopelessly naïve. They didn't understand that the simplicity of the web was actually its strength. Because linking was so straightforward, anyone could do it. That would prove to be crucial in the adoption and success of the World Wide Web.

It's all-too tempting to quickly declare that an approach is naïve, overly simplistic, and unrealistic. The idea that a website can simultaneously offer universal access to everyone while also providing a rich immersive experience for more capable devices …that also seems hopelessly naïve.

This judgement has been handed down many times over the history of the web.

“This is too simple”

When the Web Standards Project ran its campaign encouraging designers to switch from TABLEs for layout to CSS, it was met with resistance. Time and time again they were criticised for their naïvety. “Sure, a CSS-based layout might be fine for a simple personal site but there’s no way it could scale to a large complex project.”

Then Doug Bowman spearheaded the CSS-based redesign of Wired.com and Mike Davidson led the CSS-based redesign of ESPN.com. After that the floodgates opened.

When Ethan Marcotte demonstrated the power of responsive design, it was met with resistance. “Sure, a responsive design might work for a simple personal site but there’s no way it could scale to a large complex project.”

Then the Boston Globe launched its responsive site. Microsoft made their homepage responsive. The floodgates opened again.

It's a similar story today. "Sure, progressive enhancement might work for a simple personal site, but there's no way it could scale to a large complex project."

The floodgates are ready to open. We just need you to create the poster child for resilient web design.

“This is too difficult”

Building resilient websites is challenging. It takes time to apply functionality and features in a considered layered way. The payoff is a website that can better react to unexpected circumstances—unusual browsers, flaky network connections, outdated devices. Nonetheless, for many web designers, the cost in time seems to be too high.

It’s worth remembering that building with progressive enhancement doesn’t mean that *everything* needs to be made available to *everyone*. Instead it’s the core functionality that counts. If every single feature needed to be available to every browser on every device, that would indeed be an impossibly arduous process. This is why prioritisation is so important. As long as the core functionality is available using the simplest possible technology, you can—with a clear conscience—layer on more advanced features.

Still, it's entirely possible that this approach will result in duplicated work. If you build an old-fashioned client-server form submission process and then enhance it with JavaScript, you may end up repeating the form-processing on the client as well as the server. That can be mitigated if you are also using JavaScript on the server. It's theoretically possible to write universal JavaScript so that the server and browser share a single codebase. Even without universal JavaScript, I still think it's worth spending time to create technical credit.

The UK's Government Service design manual provides an even shorter form of the three-step process I've outlined:

1. First, just make it work
2. Second, make it work better

The design manual also explains why:

> *If you build pages with the idea that parts other than HTML are optional, you'll create a better and stronger web page.*

This kind of resilience means that the time you spend up-front is well invested. You might also notice an interesting trend; the more you use this process, the less time it will take.

Moving from TABLEs to CSS seemed like an impossibly idealistic goal. Designers were comfortable using TABLE and FONT elements for layout. Why would they want to learn a whole new way of working? I remember how tricky it was to make my first CSS-based layouts after years of using hacks. It took me quite some time. But my second CSS-based layout didn't take quite so long. After a while, it became normal.

Designers comfortable with fixed-width layouts had a hard time with responsive design. That first flexible layout was inevitably going to take quite a while to build. But the second flexible layout didn't take quite so long. After a while, it became normal.

It's no different with the layered approach needed for building resilient websites. If you're not used to working this way, the first time you do it will take quite some time. But the second time won't take quite so long. After a while, it will become normal.

There may well be situations where the three-step approach won't work, but they're not as common as you might think. If you're building a 3D game in a web browser, the simplest technology capable of providing the core functionality will still be pretty complex. That said, I'd love to see a text-only adventure game get enhanced into a first-person shooter.

“How do I convince…?”

The brilliant computer scientist Grace Hopper kept an unusual timepiece on her wall. It ran counter-clockwise. When quizzed on this, she pointed out that it was an arbitrary convention, saying:

> *Humans are allergic to change. They love to say, “We’ve always done it this way.” I try to fight that. That’s why I have a clock on my wall that runs counter-clockwise.*

Commodore Grace M. Hopper, United States Navy.

Behaviour change is hard. Even if you are convinced of the benefits of a resilient approach to web design, you may find yourself struggling to convince your colleagues, your boss, or your clients. It was ever thus. Take comfort from the history of web standards and responsive design. Those approaches were eventually adopted by people who were initially resistant.

Demonstrating the benefits of progressive enhancement can be tricky. Because the payoff happens in unexpected circumstances, the layered approach is hard to sell. Most people will never even know whether or not a site has been built in a resilient way. It's a hidden mark of quality that will go unnoticed by people with modern browsers on new devices with fast network connections.

For that same reason, you can start to implement this layered approach without having to convince your colleagues, your boss, or your clients. If they don't care, then they also won't notice. As Grace Hopper also said, "it's easier to ask forgiveness than it is to get permission."

Tools

Changing a workflow or a process can be particularly challenging if it clashes with the tools being used. A tool is supposed to help people get their work done in a more efficient way. The tool should be subservient to the workflow. All too often, tools instead dictate a preferred way of working. Whether it's WYSIWYG editors, graphic design programs, content management systems, or JavaScript frameworks, tools inevitably influence workflows.

If you are aware of that influence, and you can recognise it, then you are in a better position to pick and choose the tools that will work best for you. There are many factors that play into the choice of frameworks, for example: "Is it well-written?", "Does it have an active community behind it?", "Does it have clear documentation?". But perhaps the most important question to ask is, "Does its approach match my own philosophy?"

Every framework has a philosophy because every framework was written by people. If your philosophy aligns with that of the framework, then it will help you work more efficiently. But if your philosophy clashes with that of the framework, you will be fighting it every step of the way. It might even be tempting to just give up and let the framework dictate your workflow. Then the tail would be wagging the dog.

Choose your tools wisely. It would be a terrible shame if you abandoned the resilient approach to web design because of a difference of opinion with a piece of software.

Differences of opinion often boil down to a mismatch in priorities. At its heart, the progressive enhancement approach prioritises the needs of people, regardless of their technology. Tools, frameworks, and code libraries, on the other hand, are often built to prioritise the needs of designers and developers. That's not a bad thing. Developer convenience is hugely valuable. But speaking personally, I think that user needs should trump developer convenience.

When I'm confronted with a problem, and I have the choice of making it the user's problem or my problem, I'll make it my problem every time. That's my job.

Future friendly

In September of 2011, I was speaking at a conference in Tennessee along with some people much smarter than me. Once the official event was done, we lit out for the countryside where we had rented a house for a few days. We were gathering together to try to figure out where the web was headed.

A small sample of internet-enabled devices.

We were, frankly, freaked out. The proliferation of mobile devices had changed everything. Tablets were on the rise. People were talking about internet TVs. We were hoping to figure out what the next big thing would be. Internet-enabled fridges, perhaps?

In the end, the only thing we could be certain of was uncertainty:

> *Disruption will only accelerate. The quantity and diversity of connected devices—many of which we haven't imagined yet—will explode, as will the quantity and diversity of the people around the world who use them.*

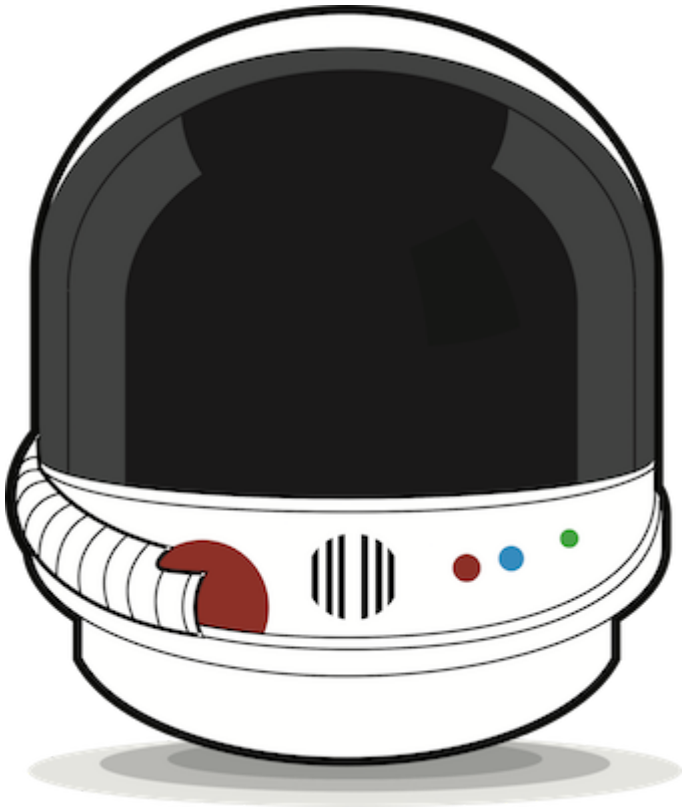
That isn't cause for despair; it's cause for celebration. We could either fight this future or embrace it. Realising that it was impossible to be future-proof, we instead resolved to be future-friendly:

1. Acknowledge and embrace unpredictability.

2. Think and behave in a future-friendly way.

3. Help others do the same.

That first step is the most important: acknowledging and embracing unpredictability. That is the driving force behind resilient web design. The best way to be future-friendly is to be backwards-compatible.



The Future Friendly symbol.

Assumptions

"We demand rigidly-defined areas of doubt and uncertainty!" cried the philosophers in Douglas Adams' *Hitchhiker's Guide To The Galaxy*.

As pattern-matching machines, we are quick to identify trends and codify them into assumptions. Here are just some assumptions that were made over the history of web design:

- Everyone has a monitor that is 640 pixels wide.
- Everyone has the Flash plugin installed.
- Everyone has a monitor that is 800 pixels wide.
- Everyone has a mouse and a keyboard.
- Everyone has a monitor that is 1024 pixels wide.
- Everyone has a fast internet connection.

The proliferation of mobile devices blew those assumptions out of the water. The rise of mobile didn't create new uncertainties—instead it shone a light on the uncertainties that already existed.

That should have been a valuable lesson. But before too long the old assumptions were replaced with new ones:

- There are some activities that people will never want to do on their phones.
- Every phone has a touch screen.
- Everyone using a phone is in a hurry.
- Every browser on every phone supports JavaScript.

These kind of assumptions always remind me of old physicist jokes. "Assume a perfectly spherical web browser…"

Assumptions are beguiling. If only we could agree on certain boundaries, then wouldn't web design be so much easier to control?

As tempting as this siren call is, it obfuscates the true nature of the ever-changing uncertain web. Carl Sagan put it best in his book *The Demon-Haunted World*:

It is far better to grasp the universe as it really is than to persist in delusion, however satisfying and reassuring.

The future

I wish I could predict the future. The only thing that I can predict for sure is that things are going to change.

I don't know what kind of devices people will be using on the web. I don't know what kind of software people will be using on the web.

The future, like the web, is unknown.

The future, like the web, will be written by you.

About the author

My name is Jeremy Keith.

I make websites at Clearleft, the design agency I co-founded in 2005. My online home is adactio.com where I've been jotting down my thoughts for over fifteen years.

Photograph by Dan Rubin.

Before writing *Resilient Web Design* I wrote the books *DOM Scripting*, *Bulletproof Ajax*, and *HTML5 For Web Designers*.

My email address is jeremy@adactio.com if you'd like to drop me a line.